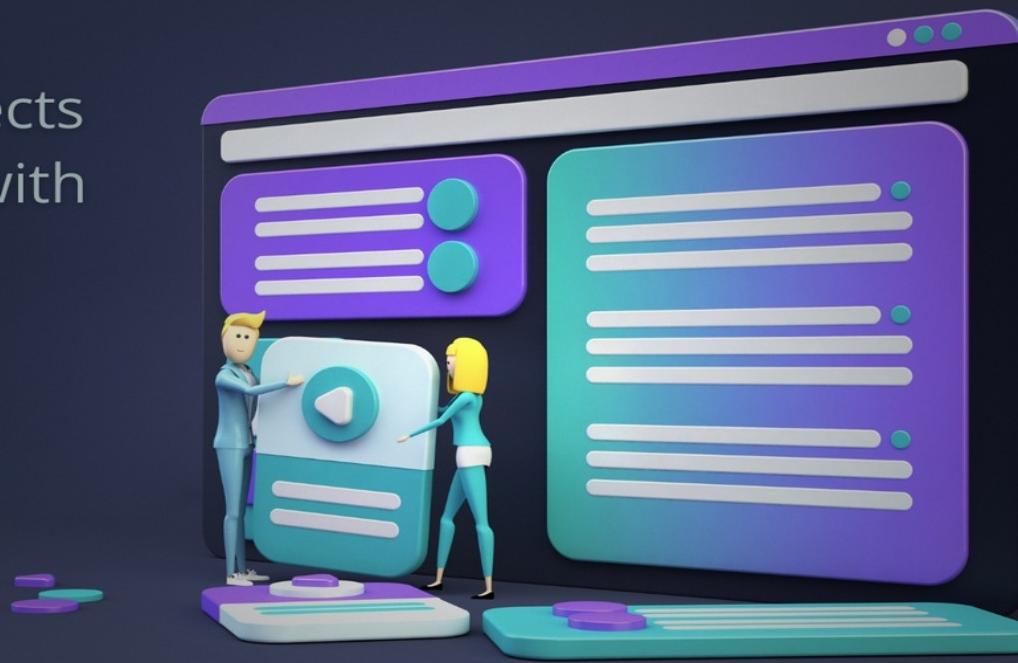


15 Web Projects With Vanilla JavaScript

Build 15 mini frontend projects from scratch with HTML5, CSS & JavaScript.



Contents

Section 1: Introduction

Chapter 1: Welcome To The Course

Chapter 2: Getting Setup

Section 2: Project 1 - Form Validator | Intro Project

Chapter 3: Project Intro

Chapter 4: Project HTML

Chapter 5: Project CSS

Chapter 6: Adding Simple Validation

Chapter 7: Check Required & Refactor

Chapter 8: Check Length, Email & Password Match

Section 3: Project 2 - Movie Seat Booking | DOM & Local Storage

Chapter 9: Project Intro

Chapter 10: Project HTML

Chapter 11: Project CSS

Chapter 12: Selecting Movie & Seats From UI

Chapter 13: Save Data To Local Storage

Chapter 14: Populate UI With Saved Data

Section 4: Project 3 - Custom Video Player | HTML5 Video API

Chapter 15: Project Intro

Chapter 16: Project HTML

Chapter 17: Project CSS

Chapter 18: Play, Pause & Stop

Chapter 19: Video Progress Bar & Timestamp

Section 5: Project 4 - Exchange Rate Calculator | Fetch & JSON Intro

- Chapter 20: Project Intro
- Chapter 21: Project HTML
- Chapter 22: Project CSS
- Chapter 23: A Look at JSON & Fetch
- Chapter 24: Fetch Rates & Update DOM

Section 6: Project 5 - DOM Array Methods | forEach, map, filter, sort, reduce

- Chapter 25: Project Intro
- Chapter 26: Project UI
- Chapter 27: Generate Random Users - Fetch w/ Async/Await
- Chapter 28: Output Users - forEach() & DOM Methods
- Chapter 29: Double Money - map()
- Chapter 30: Sort By Richest - sort()
- Chapter 31: Show Millionaires - filter()
- Chapter 32: Calculate Wealth - reduce()

Section 7: Project 6 - Menu Slider & Modal | DOM & CSS

- Chapter 33: Project Intro
- Chapter 34: Project HTML
- Chapter 35: Navbar Styling
- Chapter 36: Header & Modal Styling
- Chapter 37: Menu & Modal Toggle

Section 8: Project 7 - Hangman Game | DOM, SVG, Events

- Chapter 38: Project Intro
- Chapter 39: Draw Hangman With SVG
- Chapter 40: Main Styling

Chapter 41: Popup & Notification Styling

Chapter 42: Display Words Function

Chapter 43: Letter Press Event Handler

Chapter 44: Wrong Letters & Play Again

Section 9: Project 8 - Meal Finder | Fetch & MealDB API

Chapter 45: Project Intro

Chapter 46: Project HTML & Base CSS

Chapter 47: Search & Display Meals From API

Chapter 48: Show Single Meal Page

Chapter 49: Display Random Meal & Single Meal CSS

Section 10: Project 9 - Expense Tracker | Array Methods & Local Storage

Chapter 50: Project Intro

Chapter 51: Project HTML

Chapter 52: Project CSS

Chapter 53: Show Transaction Items

Chapter 54: Display Balance, Income & Expense

Chapter 55: Add & Delete Transactions

Chapter 56: Persist To Local Storage

Section 11: Project 10 - Infinite Scroll Posts | Fetch, Async/Await, CSS Loader

Chapter 57: Project Intro

Chapter 58: Project HTML

Chapter 59: Project CSS & Loader Animation

Chapter 60: Get Initial Posts From API

Chapter 61: Add Infinite Scrolling

Chapter 62: Filter Fetched Posts

Section 12: Project 11 - Speech Text Reader | Speech Synthesis

Chapter 63: Project Intro

Chapter 64: HTML & Output Speech Boxes

Chapter 65: Project CSS

Chapter 66: Get Speech Voices

Chapter 67: Speech Buttons

Chapter 68: Change Voice & Custom Text Box

Section 13: Project 12 - Relaxer App | CSS Animations, setTimeout

Chapter 69: Project Intro

Chapter 70: Creating The Large Circle

Chapter 71: Creating & Animating The Pointer

Chapter 72: Breath Animation With JS Trigger

Section 14: Project 13 - New Year Countdown | DOM, Date & Time

Chapter 73: Project Intro

Chapter 74: Landing Page HTML & Styling

Chapter 75: Create The Countdown

Chapter 76: Dynamic Year & Spinner

Section 15: Project 14 - Sortable List | Drag & Drop API

Chapter 77: Project Intro

Chapter 78: Insert List Items Into DOM

Chapter 79: Scramble List Items

Chapter 80: Core CSS

Chapter 81: Drag & Drop Functionality

Chapter 82: Check Order

Section 16: Project 15 - Breakout Game | HTML5 Canvas API

Chapter 83: Project Intro
Chapter 84: Creating & Styling The Page
Chapter 85: Canvas Plan Outline
Chapter 86: Draw Ball, Paddle & Score
Chapter 87: Creating The Bricks
Chapter 88: Move Paddle
Chapter 89: Move Ball & Break Bricks
Chapter 90: Lose & Reset Game
~ Conclusion

Section 1:

Introduction

Welcome To The Course

First and foremost, let me express my sincere gratitude for picking up “15 Web Projects With Vanilla JavaScript.” This coursebook is your gateway to stepping into the world of web development, where you’ll be sculpting projects using the fundamental tools of the web: HTML5, CSS, and JavaScript. As you turn the pages of this coursebook, you will embark on an immersive journey, one that promises not only to equip you with the necessary skills but also to offer an engaging hands-on experience.

What Awaits You?

Over the next several sections, we will delve deep into a series of projects, each curated to provide a balanced mixture of theory and practice. From creating form validators and custom video players to concocting

games like Hangman and Breakout, every project is designed with two primary objectives:

1. Skill Acquisition: Ensuring you gain a concrete understanding of core web development concepts.
 2. Skill Application: Enabling you to apply the acquired knowledge in real-world scenarios.
-

Why “Vanilla” JavaScript?

In the vast ocean of web development, numerous frameworks and libraries promise faster development and nifty features. While these are excellent tools for seasoned developers, it's crucial for beginners to grasp the fundamental, raw power of plain JavaScript. By focusing on “vanilla” JavaScript, we ensure that you understand the core of web scripting, making it easier for you to adapt to any library or framework in the future.

Who Is This Coursebook For?

This coursebook caters to a wide audience:

- Beginners who have dabbled a bit in HTML, CSS, and JS and are eager to build projects.
 - Intermediate developers looking to solidify their understanding and gain more hands-on experience.
 - Educators and Instructors seeking a structured curriculum for teaching web development.
-

How To Get The Most Out Of This Coursebook

1. Follow Along Actively: As this is a project-based coursebook, always have your development environment ready. Type out the code snippets, experiment with them, break them, and fix them.
2. Practice Makes Perfect: At the end of some chapters, I've included optional exercises. Engage with them to cement your understanding.

3. Stay Curious: Whenever a new concept is introduced, take a moment to explore it further. The more you challenge yourself, the better you'll grasp the nuances of web development.

Prerequisites

While this coursebook is designed to be comprehensive, having a foundational understanding of HTML, CSS, and JS will be beneficial. If you're an absolute beginner, I'd recommend starting with my "Modern HTML/CSS From The Beginning" and "Modern JS From The Beginning" courses on Udemy. This will give you a solid footing to tackle the projects in this coursebook.

Wrapping Up

Remember, every coder, no matter how skilled, started from scratch. With patience, persistence, and the right resources, you too can carve out your niche in the world of web development.

So, as we stand at the threshold of this exciting journey, take a deep breath, roll up your sleeves, and let's dive into the world of web projects with vanilla JavaScript!

Getting Setup

Welcome to the first step of our exciting journey through the world of web development! Before diving into the various projects we'll tackle, it's essential to ensure our development environment is correctly set up. This chapter will guide you through the installation and configuration of all the tools and software required to follow along with the projects in this book.

2.1. System Requirements

Before we start, ensure you have a computer with:

- An operating system like Windows, macOS, or Linux.
 - A minimum of 4GB RAM (8GB recommended for smoother performance).
 - A stable internet connection.
-

2.2. Text Editor

Every developer needs a reliable text editor. Here are some popular choices:

- Visual Studio Code (VS Code): This free, open-source editor from Microsoft is widely adopted because of its ease of use, extensive feature set, and vast library of extensions. [Download here.]
(<https://code.visualstudio.com/>)
- Sublime Text: Another favorite among developers, Sublime Text offers a clean interface and robust performance. [Download here.]
(<https://www.sublimetext.com/>)
- Atom: Developed by GitHub, Atom is a free, open-source editor that's customizable and beginner-friendly. [Download here.]
(<https://atom.io/>)

Recommended Extensions for VS Code: If you choose VS Code, consider installing extensions like `Live Server`, `Prettier`, and `ESLint` for an enhanced development experience.

2.3. Web Browsers

To view and test our projects, you'll need a modern web browser. While most browsers will work, the following are recommended due to their extensive developer tools:

- Google Chrome [Download here.]
(<https://www.google.com/chrome/>)
- Mozilla Firefox [Download here.]
(<https://www.mozilla.org/en-US/firefox/new/>)

- Microsoft Edge (Chromium version) [Download here.] (<https://www.microsoft.com/edge>)
-

2.4. Browser Developer Tools

Modern browsers come equipped with developer tools, allowing you to inspect, debug, and profile your web projects. Familiarize yourself with the developer tools of your chosen browser. They're essential for diagnosing issues and understanding how your code operates within the browser.

2.5. Node.js and npm

Some projects in this book will require the use of Node.js and npm (node package manager). Node.js allows you to run JavaScript outside the browser, while npm is the world's largest software registry.

1. Installing Node.js: Visit the [official Node.js website] (<https://nodejs.org/>) and download the LTS (Long Term Support) version. The installation process is straightforward.
2. Verifying Installation: Once installed, open your terminal or command prompt and run the following commands to ensure both Node.js and npm are installed:

```
“bash
```

```
node -v
```

```
npm -v
```

```
”
```

2.6. Version Control with Git

Git is a distributed version control system that helps track changes in your code. It's highly recommended for every developer.

1. Installing Git: Download and install Git from the [official website](<https://git-scm.com/>).
2. Verifying Installation: In your terminal or command prompt, type:

```
“bash  
git —version  
“
```

2.7. Directory Structure

Maintain a clean directory structure for your projects. Here's a suggested structure:

```
“  
/web-projects  
    /project-1-form-validator  
        /css  
        /js  
        index.html  
    /project-2-movie-seat-booking  
        /css  
        /js  
        index.html  
    ... and so on  
“
```

By keeping your files organized, you ensure that as your projects grow, you won't get lost in a maze of files and folders.

2.8. Wrapping Up

Congratulations! You've set up your development environment. As you proceed through the book, you'll gain hands-on experience and deepen your

understanding of the tools we've discussed in this chapter. Remember, web development is a journey, and every project will enhance your skills and knowledge.

Now that we're all set, let's dive into our first project in the next chapter!

Section 2: Project 1 - Form Validator | Intro Project

Project Intro

Welcome to our very first project of this exciting journey: The Form Validator! Forms are an integral part of the web, and no matter how the landscape of web development changes, form validations will always be a necessary skill for web developers. Whether you're building a sign-up page, a login form, or any other input system, ensuring the accuracy and correctness of data is vital. This introductory project is designed to provide a strong foundation for those validations.

Why Form Validation?

In the vast world of web development, why did we choose to start with form validation? Here are a few reasons:

1. **Universality:** Nearly every website you visit has some form - be it a search bar, a login page, or a feedback form.
2. **User Experience:** A good form validation system provides immediate feedback to users, ensuring they provide the right information.
3. **Security:** Ensuring data is validated on the front-end can also be a first line of defense against malicious inputs.

What We'll Build

We're going to design a simple form that asks users for a username, email, and password. This form will have the following features:

- Fields will be checked to ensure they aren't left empty.
- Email addresses will be validated to ensure they're in a proper format.
- Passwords will be validated for length and to ensure a confirm password matches.

Technologies & Techniques

Although the main focus here is on JavaScript, HTML and CSS will play crucial roles. Here's a glimpse of what we'll use:

1. HTML: Structure our form and inputs.
2. CSS: Stylish visual cues to indicate errors or successful inputs.
3. JavaScript: The heart of our validation logic. We'll be using vanilla JavaScript, without any libraries or frameworks.

Prerequisites

You should have a basic understanding of HTML, CSS, and JavaScript. If you're completely new or need a refresher, it's advisable to first check out the "Modern HTML/CSS From The Beginning" and "Modern JS From The Beginning" courses on Udemy.

What You'll Gain

By the end of this project, you'll:

1. Understand how to intercept form submissions using JavaScript.

2. Learn how to traverse and manipulate the DOM to highlight errors.
3. Be familiar with basic string methods and regular expressions for validation.
4. Have a foundational project in your portfolio to showcase basic front-end validation skills.

Let's Get Started!

Excited? You should be! This foundational project will set the pace for the other, more complex projects we'll tackle in this course. So, buckle up, and let's dive into the world of form validation!

In the next chapter, we'll begin with the structure of our form by creating its HTML layout. Let's begin this amazing journey together!

Project HTML

Welcome to Chapter 4, where we're going to dive into the heart of our first project: the Form Validator. Before we begin styling and adding dynamic functionality, we need a solid foundation. That foundation is our HTML structure.

Understanding the Project:

Before diving into the code, let's grasp the core concept of our project. A Form Validator ensures that the data entered by users meets specific criteria. This is important not only for user experience but also for security reasons.

Our form will require:

- A username
- An email address

- A password
- A password confirmation

Each of these fields will have its own validation criteria, which we'll implement in the subsequent chapters.

Setting Up the Base HTML:

Begin with the foundational structure of an HTML document:

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Form Validator</title>
    <!-- Your styles will be linked here later -->
</head>
<body>
</body>
</html>
``
```

Building the Form:

Inside the `<body>`, we'll structure our form:

```
``html
<div class="container">
    <form id="form" class="form">
        <h2>Register With Us</h2>
```

```
<!-- Username Field -->
<div class="form-control">
    <label for="username">Username</label>
    <input type="text" id="username"
placeholder="Enter username">
    <!-- Error message will be displayed here -->
    <small>Error message</small>
</div>

<!-- Email Field -->
<div class="form-control">
    <label for="email">Email</label>
    <input type="email" id="email" placeholder="Enter
email">
    <small>Error message</small>
</div>

<!-- Password Field -->
<div class="form-control">
    <label for="password">Password</label>
    <input type="password" id="password"
placeholder="Enter password">
    <small>Error message</small>
</div>

<!-- Password Check Field -->
<div class="form-control">
    <label for="password2">Confirm
Password</label>
    <input type="password" id="password2"
placeholder="Enter password again">
    <small>Error message</small>
</div>
```

```
<!-- Submit Button -->
<button type="submit">Submit</button>
</form>
</div>
"
```

Here's what we've done:

1. Container: All our form elements are wrapped within a `<div>` container to help with styling and centering the form on the page.
2. Form Controls: Each input field and its label are enclosed within a `form-control` div. This setup will help us in styling and showing error messages related to each field.
3. Error Messages: A `<small>` tag is used to display error messages. Initially, it contains a generic error message, but this will change dynamically as we validate the form.
4. Submit Button: This button will be used to trigger our validation. When the user presses it, the form will either submit (if everything is valid) or show relevant error messages.

Conclusion:

The HTML structure for our Form Validator is set up. As simple as it might look now, this structure is the backbone of our dynamic functionality.

Always remember, the key to creating effective web projects is to maintain a clean and understandable structure in the HTML, which makes styling and scripting a lot smoother. Onward to the next chapter!

Project CSS

Welcome to the styling portion of our first project: the Form Validator. As you may recall, the aim of this project is to introduce you to the basics of web development, and one of the essential aspects of building a visually appealing web application is styling. CSS (Cascading Style Sheets) allows us to apply styles to our HTML documents, making them more aesthetically pleasing and user-friendly.

For our form validator, we're going to focus on creating a neat, simple, and intuitive design that guides users through the process of inputting their details. The goal here is not just to make things look nice but also to use styles as a tool for better usability.

1. Base Styles:

Let's begin by adding some general styles to our form:

```
``css
body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
    background-color: f4f4f4;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
}
.container {
    background-color: fff;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.3);
    width: 400px;
```

```
}
```

```
"`
```

These base styles will provide a pleasant gray background, center our form on the page, and give the form container a nice shadow and rounded corners.

2. Form Styles:

Now, let's style the form elements.

```
"`css
```

```
form {  
    width: 100%;  
}  
  
form label {  
    display: block;  
    margin-bottom: 5px;  
    font-weight: bold;  
}  
  
form input[type="text"],  
form input[type="password"],  
form input[type="email"] {  
    width: 100%;  
    padding: 10px;  
    margin-bottom: 10px;  
    border: 1px solid ccc;  
    border-radius: 4px;  
}  
  
form button {  
    display: block;  
    width: 100%;
```

```
padding: 10px;  
border: none;  
border-radius: 4px;  
background-color: #333;  
color: #fff;  
cursor: pointer;  
}  
“
```

These styles make our form inputs stretch to take up the full width of the container, making it easier for users to click into them. They also apply some padding for better visual spacing and a neat appearance.

3. Validation Feedback Styles:

To provide immediate feedback to users, let's add some styles to highlight invalid and valid fields:

```
“css  
.invalid {  
    border-color: red;  
    background-color: #ffe6e6;  
}  
.valid {  
    border-color: green;  
    background-color: #e6ffe6;  
}  
.error-message {  
    color: red;  
    font-weight: bold;  
    font-size: 0.8rem;
```

```
    margin-top: -10px;  
    margin-bottom: 10px;  
}  
“
```

The `invalid` and `valid` classes provide a visual cue about the input field status by changing border colors. The `error-message` class will be used for displaying specific error messages next to our input fields.

4. Responsive Design:

Considering the growing number of mobile users, our form should be responsive:

```
“css  
@media screen and (max-width: 420px) {  
    .container {  
        width: 90%;  
    }  
}
```

This media query ensures that on smaller screens, our form takes up most of the viewport width, leaving a small margin on the sides.

Conclusion:

CSS is a powerful tool that goes beyond just making things look good. With the styles we've added, our form is now not only visually appealing but also user-friendly. The visual cues from our validation feedback styles will guide users through the form, making the entire process intuitive and efficient.

Adding Simple Validation

In the world of web development, one thing you'll come across often is the need to validate forms. Whether it's a sign-up form, a login page, or a settings update page, it's essential to ensure that the data users provide is correct and secure. In this chapter, we will delve deep into creating simple form validation for our introductory project using vanilla JavaScript.

Why is validation important?

Before diving into the code, let's understand why validation is crucial:

1. User Experience (UX): Proper validation ensures users fill out forms correctly, reducing errors and misunderstandings.
 2. Security: Validating inputs helps prevent malicious users from sending harmful data to your server.
 3. Data Integrity: By validating forms, you ensure that the data sent to the server adheres to the expected format.
-

Getting Started with Simple Validation

For our intro project, we have a basic form that we want users to fill out. Let's assume it contains fields like `username`, `email`, and `password`. Our task is to validate these fields using simple checks.

HTML Structure:

```
“html
<form id=“userForm”>
    <input type=“text” id=“username”
placeholder=“Username”>
    <input type=“email” id=“email” placeholder=“Email”>
    <input type=“password” id=“password”
placeholder=“Password”>
    <button type=“submit”>Submit</button>
```

```
<p id="error-message"></p>
</form>
```

“

The `error-message` paragraph is where we will display any validation errors to the user.

JavaScript Validation:

Begin by selecting our form and adding an event listener for the `submit` event.

```
``javascript
const form = document.getElementById('userForm');
form.addEventListener('submit', function(e) {
    e.preventDefault(); // Prevents the form from
    // submitting
    validateForm();
});
```

“

Next, we'll define our `validateForm` function:

```
``javascript
function validateForm() {
    const username =
        document.getElementById('username').value;
    const email =
        document.getElementById('email').value;
    const password =
        document.getElementById('password').value;
    const errorMessage =
        document.getElementById('error-message');

    // Resetting the error message
    errorMessage.textContent = "";
```

```
// Validate Username
if(username === "" || username.length < 3) {
    errorMessage.textContent += 'Username must be
at least 3 characters long.\n';
    return;
}

// Validate Email
const emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-
9.-]+.[a-zA-Z]{2,6}$/;
if(!emailPattern.test(email)) {
    errorMessage.textContent += 'Please enter a valid
email address.\n';
    return;
}

// Validate Password
if(password === "" || password.length < 8) {
    errorMessage.textContent += 'Password must be
at least 8 characters long.\n';
    return;
}
``
```

Here's a brief overview of our validation checks:

- Username: It shouldn't be empty, and its length should be at least 3 characters.
 - Email: We use a regular expression (regex) to ensure the email is in the correct format.
 - Password: It shouldn't be empty, and its length should be at least 8 characters.
-

Conclusion

Form validation is essential for ensuring user data is accurate, safe, and user-friendly. With just vanilla JavaScript, you can put simple validations in place to enhance the user experience and protect your application. As you delve deeper into more complex projects, you'll encounter scenarios that require more intricate validation rules. Still, the foundation remains the same: ensuring users provide data that aligns with your application's expectations.

In the next chapter, we'll enhance our validation process by refactoring our code and introducing more validation checks. Stay tuned!

Check Required & Refactor

In this chapter, we'll delve into the core of form validation: checking for required fields. It's essential that users provide the necessary information when filling out forms, so ensuring that they've filled out all required fields is a fundamental step. But simply checking isn't enough; our code should be clean and maintainable. So, we'll also spend some time refactoring our code to make it efficient and easily readable.

Understanding Required Fields

A required field is any input within your form that must contain a value before the form can be successfully submitted. For example, in a registration form, fields like "Username," "Password," and "Email" are typically marked as required.

Steps for Checking Required Fields:

1. Selection: Select all the input fields that you want to validate.

2. Checking: For each input field, check if its value is empty.

3. Feedback: If the value is empty, provide feedback to the user indicating that the field is required.

Coding the Required Fields Check:

First, let's select all the input fields:

```
``javascript
const form = document.getElementById('register-form');

const username =
document.getElementById('username');

const email = document.getElementById('email');

const password =
document.getElementById('password');

const password2 =
document.getElementById('password2');

``
```

Now, let's create a function to check required fields:

```
``javascript
function checkRequired(inputArr) {
    inputArr.forEach(function(input) {
        if (input.value.trim() === "") {
            showError(input, `${getFieldName(input)} is required`);
        } else {
            showSuccess(input);
        }
    });
}
```

```
function getFieldName(input) {  
    return input.id.charAt(0).toUpperCase() +  
    input.id.slice(1);  
}  
“
```

In the `checkRequired` function, we're looping through each input field to check if its value is empty. If it is, we display an error; otherwise, we indicate success.

The `getFieldName` function is a utility function to get a more readable name for our input field. This function capitalizes the first letter of the input's ID for use in our error message.

Now, let's invoke our `checkRequired` function:

```
“javascript  
form.addEventListener('submit', function(e) {  
    e.preventDefault();  
    checkRequired([username, email, password,  
    password2]);  
});  
“
```

Refactoring the Code

Refactoring involves restructuring existing code without changing its functionality. It's all about making the code more efficient, readable, and maintainable.

Considering our form validator, here are some steps to refactor our code:

1. Avoid Repetition: We already began this by creating the `checkRequired` function. By packaging the repeated logic inside this function, we can simply call it when needed rather than writing the same logic over and over.

2. Utility Functions: Breaking down complex tasks into smaller utility functions, like our `getFieldName`, can make the code much more readable. Each utility function should perform a specific task.
3. Comments: While our code should be self-explanatory, adding a few comments explaining complex or crucial parts can be very helpful for others (or even for ourselves in the future).
4. Consistent Naming: Ensuring that we have a consistent naming convention can make the code much more readable.

Conclusion

Checking required fields is a fundamental step in form validation. With our `checkRequired` function, we can now easily ensure that users have filled out all essential fields. Moreover, by refactoring our code, we've made it more efficient and maintainable. As we progress with our form validator project, remember the importance of clean code. It's not just about getting it to work; it's about building something that's robust and easy to understand.

In the next chapter, we will further enhance our validator by adding checks for input length, email format, and password match. Stay tuned!

Check Length, Email & Password Match

Welcome to the next chapter of our Form Validator Intro Project! In the previous chapter, we went over how to add simple validation checks. Now, we'll dive deeper into the specifics by implementing checks for length constraints, email format, and matching passwords. Let's jump in!

Understanding the Importance of Validations

Before delving into the code, it's important to recognize why these checks are crucial. By ensuring the length of inputs like username and password, we add a layer of security and consistency to our forms. Validating email formats ensures users provide a legitimate email, and checking for matching passwords confirms user intent.

Check Input Length

To ensure that our users have a username and password of a certain length, we'll set minimum and maximum limits.

JavaScript:

```
``javascript
function checkLength(input, min, max) {
    if (input.value.length < min || input.value.length >
max) {
        showError(input, `${getFieldName(input)} must be
between ${min} and ${max} characters`);
    } else {
        showSuccess(input);
    }
}```
```

Here, the `checkLength` function accepts an input element and checks its value's length against the given minimum and maximum lengths. If it doesn't meet these criteria, it displays an error message.

Validate Email Format

To ensure our users provide a valid email address, we'll use a regular expression to validate the format.

JavaScript:

```
``javascript
function isValidEmail(input) {
    const re = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,6}$/;
    if (re.test(input.value.trim())) {
        showSuccess(input);
    } else {
        showError(input, 'Email is not valid');
    }
}
``
```

Regular expressions can be tricky, but the above pattern checks for common characteristics of valid email addresses.

Check Password Match

Matching passwords ensures that users have entered their intended password correctly in both fields.

JavaScript:

```
``javascript
function checkPasswordsMatch(input1, input2) {
    if (input1.value !== input2.value) {
        showError(input2, 'Passwords do not match');
    }
}
``
```

This function compares the values of two input fields. If they don't match, an error is displayed.

Bringing It All Together

Now, let's implement these validation checks in our form's submit event:

```
``javascript
form.addEventListener('submit', function(e) {
    e.preventDefault();
    checkLength(username, 3, 15);
    checkLength(password, 6, 25);
    isValidEmail(email);
    checkPasswordsMatch(password, password2);
});``
```

When our form is submitted, these validation checks will be executed, ensuring our form data is both secure and consistent.

Conclusion

By the end of this chapter, you've learned to implement specific validation checks that are common in many web applications. Ensuring the proper length, format, and consistency of user input is essential for both usability and security. As we progress through the course, you'll discover more advanced techniques to enhance your form validation processes further.

Section 3: Project 2 - Movie Seat Booking | DOM & Local Storage

Project Intro

Welcome to Project 2, where we'll dive deep into the powerful capabilities of the Document Object Model (DOM) and the wonders of the Local Storage API. By the end of this project, you'll have built a fully functional movie seat booking application, an asset to add to your growing web development portfolio.

Overview

Imagine a world where every time you wanted to watch a movie, you'd simply load up your own app, select the movie, choose your preferred seats, and voila! The app would remember your choice, even if you accidentally closed it. Sounds great, right? That's precisely what we're building in this project.

Key Features

1. Dynamic Movie Selection: Display a list of movies with their prices and allow users to choose one.
 2. Interactive Seat Map: A graphical representation of the movie theater, where users can click on individual seats to select or deselect them.
 3. Seat Booking: Allow users to book their desired seats by clicking on them. Booked seats will have a different color or style to distinguish them from available seats.
 4. Price Calculation: Display the total price based on the number of seats selected and the price of the chosen movie.
 5. Local Storage Capabilities: Save the user's movie and seat selection even if they close the browser. When they revisit, their choices should still be there.
-

What You Will Learn

- DOM Manipulation: Learn how to interact with and modify web page elements in real-time, making your web applications interactive and dynamic.

- Event Handling: Detect and respond to user actions, such as clicking on a seat or selecting a movie.
 - Using Local Storage: Store user data, such as their seat selection, on their own computer, ensuring that their preferences remain saved even after closing the browser.
 - Data Retrieval & Display: Retrieve data from Local Storage and display it appropriately when the user revisits the page.
-

Real-World Applications

Beyond just being a fun project, understanding these concepts is crucial for building interactive web applications.

- E-commerce sites use similar techniques for shopping carts, ensuring that your selected items remain in the cart even after navigating away.
- Forms on many websites remember your input if you navigate away and come back, enhancing user experience.

By mastering these concepts here, you'll be well-equipped to tackle real-world challenges in web development.

Prerequisites

Before diving into this project, it would be beneficial to have:

1. A basic understanding of HTML, CSS, and JavaScript. If you're new to these, refer back to our foundational chapters or courses.
 2. Familiarity with the basics of the DOM, particularly how to select and modify elements.
-

Conclusion

This project is an exciting blend of front-end design with backend-like functionalities using pure front-end technologies. Not only will it boost your confidence in building real-world projects, but it will also be an excellent addition to your portfolio, showcasing your prowess in DOM manipulation and Local Storage.

Project HTML

Welcome to the heart of our second project: the Movie Seat Booking application. This project will give you practical experience working with the Document Object Model (DOM) and the Local Storage of a browser. But before we delve into the interactive JavaScript and CSS styling, we need to lay the groundwork with our HTML. This is where it all starts.

1. Setting Up Our HTML Document

To begin, let's set up our basic HTML structure:

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Movie Seat Booking</title>
    <link rel="stylesheet" href="styles.css">
    <script src="script.js" defer></script>
</head>
<body>
</body>
```

```
</html>
```

```
"`
```

This structure references an external stylesheet (`styles.css`) and an external JavaScript file (`script.js`). The `defer` attribute ensures our JavaScript won't run until the HTML document is fully loaded.

2. The Main Container

Inside the body, our main content will reside within a centered container:

```
"`html
<div class="container">
    <!-- Content goes here -->
</div>
"`
```

3. Adding a Title and Movie Selection

First, let's provide a title for our movie theater, followed by a dropdown menu for movie selection:

```
"`html
<h2>Select a Movie:</h2>
<select id="movie">
    <option value="10">Avengers: Endgame ($10)
</option>
    <option value="8">Joker ($8)</option>
    <option value="12">Frozen II ($12)</option>
    <option value="9">Toy Story 4 ($9)</option>
</select>
"`${
```

The `value` attribute for each movie represents its ticket price.

4. Theater Layout

For our theater layout, we'll represent seats using div elements. Unoccupied seats will be clickable, allowing users to choose them. Once selected, the seat's status will be updated to show that it's occupied.

```
``html
```

```
<div class="cinema">  
    <!-- We'll use repeated divs to denote individual  
    seats -->  
    <div class="row">  
        <div class="seat"></div>  
        <div class="seat"></div>  
        <!-- Add more seats as required -->  
    </div>  
    <!-- Repeat rows as required -->  
</div>
```

```
``
```

5. Information Display

Beneath the cinema layout, we'll provide an area to display information to the user:

```
``html
```

```
<div class="info">  
    <p>Total Tickets: <span id="ticketCount">0</span>  
</p>  
    <p>Total Price: $<span id="totalPrice">0</span></p>  
</div>
```

```
``
```

The information display shows the number of tickets and the total price.

6. Conclusion

This sets the foundation for our movie seat booking app. With the HTML in place, we'll be diving into the CSS to make our app visually appealing in the next chapter, followed by the JavaScript to give it dynamic functionality.

Remember, a well-structured HTML will simplify the process when it comes to styling and adding interactivity. Think of this as building the skeleton of our application, upon which we'll layer on the muscles (CSS) and the brain (JavaScript).

In the next chapter, we'll delve into the styling of this project. Make sure your HTML structure aligns with what we've covered here, as it will be crucial for our CSS and JavaScript to function correctly.

Project CSS

In this chapter, we'll be diving deep into styling our Movie Seat Booking App using pure CSS. Our goal is to create an intuitive, engaging, and aesthetically pleasing user interface that aligns well with the functionality we will develop in the subsequent chapters.

CSS Structure:

We will break down our styles into three main parts:

1. Base Styles: This is where we set up our foundational styles such as the app's font, colors, and general layout.
2. Component Styles: These will include styles for specific components like buttons, seat selections, and movie dropdown.
3. Utility Styles: General utility styles that might be reused.

1. Base Styles

HTML Body and Container:

```
``css
body {
    font-family: 'Arial', sans-serif;
    background-color: f4f4f4;
    margin: 0;
    padding: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
}

.container {
    background-color: fff;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    width: 80%;
    max-width: 800px;
}
```

``

Default Text Styles:

```
``css
h1, h2, h3 {
    margin-top: 0;
}

p {
```

```
    margin-bottom: 20px;  
}  
“
```

2. Component Styles

Movie Dropdown:

```
“css  
.movie-select {  
    display: flex;  
    justify-content: space-between;  
    margin: 20px 0;  
}  
.movie-select select {  
    padding: 10px 15px;  
    border-radius: 5px;  
    border: 1px solid e0e0e0;  
}  
“
```

Seats in Cinema:

To visualize the cinema, we will use a flexbox to position the seats and the screen.

```
“css  
.cinema {  
    perspective: 1000px;  
}  
.screen {  
    background-color: d3d3d3;  
    height: 70px;  
    width: 100%;
```

```
margin: 20px 0;  
text-align: center;  
line-height: 70px;  
font-size: 1.5em;  
letter-spacing: 3px;  
}  
.seats {  
display: flex;  
flex-direction: row;  
flex-wrap: wrap;  
position: relative;  
}  
.seat {  
background-color: BDC3C7;  
width: 20px;  
height: 20px;  
margin: 5px;  
border-top-left-radius: 10px;  
border-top-right-radius: 10px;  
cursor: pointer;  
}  
.seat.occupied {  
background-color: E74C3C;  
cursor: not-allowed;  
}  
.seat.selected {  
background-color: 2ECC71;  
}
```

```
```
Summary and Button Styles:
``css
.summary {
 margin-top: 20px;
 display: flex;
 justify-content: space-between;
}
button {
 padding: 10px 20px;
 border: none;
 border-radius: 5px;
 background-color: 2ECC71;
 color: ffffff;
 cursor: pointer;
 transition: background-color 0.3s;
}
button:hover {
 background-color: 27AE60;
}
``
```

---

### 3. Utility Styles

```
``css
.clearfix::after {
 content: "";
 clear: both;
 display: table;
```

```
}
```

```
"
```

## Conclusion

Styling is an integral part of any web application, and as we've seen in this chapter, it plays a pivotal role in the user experience. With our CSS now set up, our Movie Seat Booking application is not only functional but also visually engaging.

# Selecting Movie & Seats From UI

In this chapter, we will walk through the process of creating the core functionality of our movie seat booking application. Our main focus will be to understand and implement the logic that allows users to select a movie and its corresponding seats from the UI.

## 1. Creating the HTML Layout:

Before diving into the JavaScript, ensure that your HTML is properly set up. For this section, we'll have a dropdown list of movies with their respective prices and a layout representing the theater's seats.

```
``html
<div class="movie-container">
 <label>Choose a movie:</label>
 <select id="movie">
 <option value="10">Avengers: Endgame ($10)
 </option>
 <option value="12">Joker ($12)</option>
 <option value="8">Frozen II ($8)</option>
```

```
<option value="9">Ford v Ferrari ($9)</option>
</select>
</div>
<ul class="seating-chart">
 <!-- Each list item represents a seat -->
 <!-- Using data attributes to indicate seat status:
available, occupied -->
 <li class="seat" data-status="available">
 <!-- ... repeat for all seats in the theater -->

``
```

---

## 2. JavaScript: DOM Selection:

Start by selecting the elements from the DOM:

```
``javascript
const movieSelect = document.getElementById('movie');
const seats = document.querySelectorAll('.seating-chart
.seat[data-status="available"]');

``
```

---

## 3. Listening for Events:

Now, we want to detect when a user selects a movie or clicks on a seat:

```
``javascript
movieSelect.addEventListener('change',
updateSelectedMovie);

seats.forEach(seat => seat.addEventListener('click',
toggleSeatSelection));

``
```

---

#### 4. Toggling Seat Selection:

We want users to be able to select and deselect seats. When a seat is selected, we'll change its data attribute:

```
``javascript
function toggleSeatSelection(e) {
 const seat = e.target;
 // Toggle the data-status attribute
 if (seat.dataset.status === 'available') {
 seat.dataset.status = 'selected';
 } else {
 seat.dataset.status = 'available';
 }
 // Update a visual representation (maybe change seat
 color)
 seat.classList.toggle('selected-seat');
 // You can now also call another function to calculate
 and display total cost
 updateTotalCost();
}
``
```

---

#### 5. Updating the Selected Movie:

When the user selects a different movie, we'll store the movie's value (price) for further calculations:

```
``javascript
let ticketPrice = +movieSelect.value;
function updateSelectedMovie(e) {
 ticketPrice = +e.target.value;
 // Resetting previously selected seats when switching
 movies might be a good idea
}
```

```
 resetSelectedSeats();
 updateTotalCost();
}

```

```

6. Calculating Total Cost:

After selecting the seats and the movie, we'll provide a live update of the total cost:

```
```javascript
function updateTotalCost() {
 const selectedSeats =
 document.querySelectorAll('.seating-chart .seat[data-
status="selected"]');
 const numberOfSeats = selectedSeats.length;
 const totalCost = numberOfSeats * ticketPrice;
 // Display this to the user using DOM manipulation
 document.getElementById('total-cost').textContent =
`$${totalCost}`;
}
```

```

Remember, this chapter focuses on the front-end aspect. In real-world applications, you'd want to synchronize with a backend to ensure seat availability, especially in scenarios where multiple users might be trying to book the same seats simultaneously.

Conclusion:

By now, you should have a functional UI that allows users to select a movie, choose their preferred seats, and view the total cost of their selection. In the next chapter, we'll look into saving these details into local

storage and populating the UI with saved data to enhance the user experience.

Save Data To Local Storage

Local storage is an essential part of modern web development, especially when we want to provide a seamless user experience. In the context of our Movie Seat Booking project, it makes sense to utilize local storage to save the user's seat selection and movie choice. This way, even if they leave the page and come back, their choices remain intact.

Understanding Local Storage

Local storage is a way to store data on the user's browser. It's like a mini database in the browser that developers can use to store key-value pairs.

Advantages:

- Data is saved across browser sessions.
- No expiration time; the data remains until it's explicitly deleted or the user clears their browser data.
- Unlike cookies, local storage is not sent to the server with every HTTP request, making your application faster and more efficient.

Limitations:

- Typically, browsers give you up to 5-10MB of storage.
- It's synchronous, meaning it might block the main thread if you're trying to store a large amount of data.

Implementing Local Storage in our Movie Seat Booking Application

For our project, we'll save the selected movie and its price, as well as the indices of selected seats. This way,

when the user revisits the page, we can repopulate the UI with their selections.

Step 1: Saving the Selected Movie

Whenever a user selects a movie from the dropdown, we should save that selection to local storage.

```
``javascript
const movieSelect = document.getElementById('movie');
movieSelect.addEventListener('change', (e) => {
  const selectedMovieIndex = e.target.selectedIndex;
  const selectedMoviePrice = e.target.value;
  localStorage.setItem('selectedMovieIndex',
  selectedMovieIndex);
  localStorage.setItem('selectedMoviePrice',
  selectedMoviePrice);
});``
```

Step 2: Saving Selected Seats

To store seat selections, we'll keep track of the indices of the selected seats. These indices will represent the seats in our UI.

```
``javascript
const seats = document.querySelectorAll('.row
.seat:not(.occupied)');
seats.forEach(seat, index) => {
  seat.addEventListener('click', () => {
    // Toggle the selected class on click
    seat.classList.toggle('selected');
    // Save the updated seat selection
    saveSelectedSeats();
});``
```

```
});

function saveSelectedSeats() {
    const selectedSeats = document.querySelectorAll('.row
.seat.selected');

    const seatsIndex = [...selectedSeats].map(seat => [...seats].indexOf(seat));

    localStorage.setItem('selectedSeats',
JSON.stringify(seatsIndex));

}
```

```

Note: We're using `JSON.stringify()` because local storage can only store strings. By converting our array of seat indices to a string, we can save and retrieve it with ease.

---

## Populating the UI with Saved Data

When the user revisits our booking page, we should check local storage for any saved data and update our UI accordingly.

### Step 1: Setting the Selected Movie

```
``javascript
const selectedMovieIndex =
localStorage.getItem('selectedMovieIndex');

if (selectedMovieIndex !== null) {
 movieSelect.selectedIndex = selectedMovieIndex;
}
``
```

### Step 2: Displaying the Selected Seats

```
``javascript
const selectedSeats =
JSON.parse(localStorage.getItem('selectedSeats'));
```

```
if (selectedSeats !== null && selectedSeats.length > 0) {
 seats.forEach((seat, index) => {
 if (selectedSeats.indexOf(index) > -1) {
 seat.classList.add('selected');
 }
 });
}
``
```

---

## Conclusion

Utilizing local storage in our Movie Seat Booking project ensures a user-friendly experience. This feature is especially crucial for scenarios where a user might accidentally refresh or navigate away from the page. With local storage, their seat and movie selections remain, saving them from the hassle of reselecting.

In the next chapter, we will explore how to populate the UI with the saved data from local storage, ensuring a seamless experience for returning users.

# Populate UI With Saved Data

Welcome back! In the previous chapter, we discussed how to save the selected movie and seat data to the local storage. Now, when a user visits the movie seat booking page, they should be able to see their previously selected seats. This is essential for a user-friendly experience. In this chapter, we'll populate the user interface (UI) with the saved data from local storage.

---

## 1. Understand the Purpose

Populating the UI with saved data helps in providing a seamless user experience. When a user returns to our movie booking site, they don't have to select their seats and movie choice again; it will automatically show their previous selections.

---

## 2. Retrieve Data From Local Storage

Before we populate our UI, we must first retrieve the saved data.

```
``javascript
const selectedSeats =
JSON.parse(localStorage.getItem('selectedSeats'));
const selectedMovieIndex =
localStorage.getItem('selectedMovieIndex');
``
```

In the code above:

- We retrieve the `selectedSeats` from local storage and parse them from string format to an array using `JSON.parse()`.
  - The `selectedMovieIndex` is retrieved as is because it's stored as a string.
- 

## 3. Populate Movie Selection

To populate the movie dropdown with the saved movie choice, we use the retrieved `selectedMovieIndex`.

```
``javascript
const movieSelect = document.getElementById('movie');
if (selectedMovieIndex !== null) {
 movieSelect.selectedIndex = selectedMovieIndex;
}
``
```

Here, we simply set the `selectedIndex` property of our `movieSelect` dropdown to the retrieved index, making sure it's not null.

---

#### 4. Populate Seats

Now, let's populate the seats based on the saved selections:

```
``javascript
const seats =
document.querySelectorAll('.seat:not(.occupied)');
if (selectedSeats !== null && selectedSeats.length > 0) {
 seats.forEach((seat, index) => {
 if (selectedSeats.indexOf(index) > -1) {
 seat.classList.add('selected');
 }
 });
}
``
```

In this code:

- We first select all the seats that are not occupied.
  - We then check if `selectedSeats` contains data and loop through each seat.
  - If the seat's index is found in the `selectedSeats` array, we add the `selected` class to that seat, visually marking it as selected.
- 

#### 5. Update Count and Total Price

Remember, our UI should also reflect the correct count of selected seats and the total price. Let's make sure we handle that too.

```
``javascript
```

```
function updateSelectedCountAndTotal() {
 const selectedSeats =
document.querySelectorAll('.row .seat.selected');
 const count = selectedSeats.length;
 const total = count * +movieSelect.value;
 document.getElementById('count').innerText = count;
 document.getElementById('total').innerText = total;
}

// Call this function after populating the seats
updateSelectedCountAndTotal();
``
```

---

## 6. Final Thoughts

It's essential always to consider the user's experience when developing applications. By populating the UI with saved data, we save the user time and provide a sense of continuity, especially if they had to navigate away from the page and then come back.

In the next section, we'll dive into the custom video player and how to harness the power of the HTML5 Video API. Stay tuned!

# Section 4: Project 3 - Custom Video Player | HTML5 Video API

## Project Intro

In today's internet-driven world, video content is one of the most powerful and persuasive mediums to convey

information. Sites like YouTube, Vimeo, and other media hosting platforms thrive because of our innate love for visual storytelling. But, as a web developer, have you ever thought about the magic that goes behind that play button, the volume slider, or the fullscreen toggle? Custom video players are at the heart of this magic.

Welcome to Project 3: the Custom Video Player! This project is more than just embedding a video onto a web page. By the end of this module, you will create a video player tailored to your design, enriched with features, all using the powerful HTML5 Video API.

---

### Why a Custom Video Player?

1. Control Over Design: You're not restricted to the standard video controls. Want a vintage volume knob instead of a slider? Go for it!
  2. Enhanced Features: Add features like speed control, custom captions, or even a video bookmarking system.
  3. Integration: Seamlessly integrate the player with other systems, like analytics to track user engagement.
- 

### What Will You Learn?

- HTML5 Video Element: Dive deep into the `<video>` element, its attributes, and understand its capabilities.
  - Custom Controls: Replace the browser's default controls with your own play, pause, volume, and fullscreen buttons.
  - Video API: Harness the power of the HTML5 Video API to manipulate video playback programmatically.
  - Styling the Player: Use CSS to style your video player, making it responsive and visually appealing.
  - Events and Interactivity: Understand the different events triggered during video playback and use them to create interactive features.
-

## Project Overview

In this project, you will build a custom video player with the following features:

1. Basic Controls: Play, pause, stop, and a volume slider.
2. Playback Progress: A slider that shows video playback progress and allows users to skip to different parts.
3. Timestamp: Display the current time and total video duration.
4. Fullscreen Toggle: Allow users to switch to fullscreen mode.
5. Responsive Design: Ensure that the player looks great on all devices.

By tackling this project, you're diving into a real-world task that developers encounter when building media-rich web applications. Whether it's for a personalized portfolio, a bespoke enterprise application, or a media startup idea, mastering the custom video player is a skill that will set you apart.

---

In the subsequent chapters, we will break down the creation process into digestible sections, covering the design, functionality, and best practices. By the end, you'll have a fully functioning video player that you can be proud of, with the skills and knowledge to customize it further as you see fit. Let's dive in!

# Project HTML

Welcome to the next exciting phase of our project series! In this chapter, we'll be laying the foundation for our Custom Video Player project by setting up the HTML structure. This player will leverage the HTML5 Video API, giving us capabilities to play, pause, skip, and more.

But before any of that, we need to build our HTML scaffold!

---

### HTML5 Video Element:

HTML5 introduced the `<video>` element which allows us to embed video files that can be played directly in the browser without the need for external plugins. Alongside the video element, there are multiple attributes and child elements we can use to enhance the user experience, such as adding controls, setting a poster image, or even providing multiple sources for different video formats.

---

### Setting Up the Basic Structure:

Let's dive in and create our HTML structure for the video player.

```
“html
<!DOCTYPE html>
<html lang=“en”>
<head>
 <meta charset=“UTF-8”>
 <meta name=“viewport” content=“width=device-width,
initial-scale=1.0”>
 <title>Custom Video Player</title>
 <!-- You’ll link your CSS here -->
 <link rel=“stylesheet” href=“styles.css”>
</head>
<body>
 <div class=“video-container”>
 <video id=“video” width=“750” height=“500”
controls>
 <!-- Insert your video source(s) here -->
```

```
<source src="path-to-your-video.mp4"
type="video/mp4">
 <!-- You can provide multiple sources for different
video formats -->
 <source src="path-to-your-video.webm"
type="video/webm">
 <!-- Display text for browsers that don't support
the video tag -->
 Your browser does not support the video tag.
</video>
<div class="controls">
 <button id="play-pause">Play</button>
 <input type="range" id="volume" min="0" max="1"
step="0.1">
 <button id="full-screen">Full-Screen</button>
 00:00 / 00:00
</div>
</div>
<!-- You'll link your JavaScript here -->
<script src="script.js"></script>
</body>
</html>
``
```

### Explaining the Code:

- Video Container: This is a wrapping `div` that contains the video and its custom controls. It will help in styling and positioning.
- Video Element: We've provided two `` elements within the `` tag. This is to ensure maximum compatibility across browsers. Different browsers might support different video formats, so by

providing multiple formats, we ensure that most users can view the video.

The `controls` attribute, when present, displays the default browser controls for the video player. While we'll be building our own custom controls, this is useful during the setup phase to ensure our video is loading correctly.

- Custom Controls: Underneath the video, we have a `div` with a class of `controls`. This will contain our custom controls for the video player, including:

- Play/Pause Button: A toggle button to play or pause the video.
- Volume Control: An input range to adjust the video's volume.
- Full-Screen Button: A button to toggle full-screen mode.
- Time Display: A span to display the current time of the video and its total duration.

---

### Conclusion:

Setting up the HTML is the first step in creating our Custom Video Player. This structure provides a foundation upon which we'll build our styles and functionalities. In the upcoming chapters, we will focus on styling this structure with CSS and then implementing the video controls with JavaScript. The goal is to create a player that is both functional and aesthetically pleasing, offering users an enhanced video playback experience.

## Project CSS

Welcome back! Now that we've set up the HTML structure for our custom video player in the previous chapter, it's time to style it and give it an appealing look. CSS is the backbone of web aesthetics. In this chapter,

we'll focus on making our video player not just functional, but also user-friendly and visually appealing.

---

### 1. Basic Reset:

To begin with, let's reset some default styles. This will help us maintain consistency across browsers.

```
``css
* {
 margin: 0;
 padding: 0;
 box-sizing: border-box;
}
``
```

---

### 2. Setting Up the Container:

Our video player will sit inside a container. This container will hold the video and the control bar.

```
``css
.video-container {
 width: 80%;
 max-width: 800px;
 margin: 40px auto;
 position: relative;
 box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}
``
```

---

### 3. Styling the Video Element:

The `<video>` element will take up the entire width of its container. We'll also give it a fixed height.

```
``css
video {
 width: 100%;
 height: 450px;
 display: block;
}
``
```

---

#### 4. Video Controls:

Below the video, we'll have our custom controls which will include play/pause buttons, a progress bar, volume control, and full-screen toggling.

```
``css
.video-controls {
 display: flex;
 align-items: center;
 justify-content: space-between;
 background-color: #333;
 color: #fff;
 padding: 10px 15px;
}
``
```

##### 4.1 Buttons:

All buttons will have a consistent style.

```
``css
.video-controls button {
 background: none;
 border: none;
 color: #fff;
```

```
 font-size: 16px;
 margin-right: 15px;
 cursor: pointer;
}
“
```

For any active or hover state:

```
“css
.video-controls button.active,
.video-controls button:hover {
 color: ff6347;
}
“
```

#### 4.2 Progress Bar:

The progress bar will show how much of the video has been played.

```
“css
.progress-bar {
 flex: 1;
 height: 5px;
 background-color: 555;
 margin: 0 10px;
 position: relative;
}
“
```

For the played section:

```
“css
.progress-bar span {
 display: block;
```

```
height: 100%;
background-color: ff6347;
position: absolute;
left: 0;
top: 0;
width: 0% /* This will be updated dynamically using
JavaScript */
}
“
```

#### 4.3 Volume Control:

```
“css
.volume {
 display: flex;
 align-items: center;
}
.volume-slider {
 width: 100px;
 margin-left: 10px;
}
“
```

---

#### 5. Full Screen Styling:

When in full screen mode, we want our video to occupy the entire viewport.

```
“css
video:-webkit-full-screen {
 width: 100vw;
 height: 100vh;
}
```

```
video:fullscreen {
 width: 100vw;
 height: 100vh;
}
“
```

That wraps up our styling for the custom video player. As you can see, with just a bit of CSS, we've transformed our raw HTML structure into a visually appealing and user-friendly video player. In the next chapter, we'll dive into making this player functional by using the HTML5 Video API and JavaScript.

*Note: Always remember to test your styles across different browsers to ensure compatibility and consistent appearance. Adjustments might be necessary based on browser-specific quirks.*

## Play, Pause & Stop

Welcome back, dear readers! In this chapter, we are going to dive deep into the core functionalities of our Custom Video Player: play, pause, and stop. These are basic yet crucial controls for any media player, and with the help of the HTML5 Video API, implementing them is quite straightforward.

---

### Introduction to HTML5 Video API:

Before diving into the coding section, it's essential to understand the power of the HTML5 Video API. This API provides a way to programmatically access and manipulate video content played back through the `<video>` element in web pages. It exposes methods and properties that allow developers to play, pause, adjust volume, and even check if a video has ended.

For our custom video player, the primary methods we'll focus on are:

1. `play()`: Initiates playback of the video.
2. `pause()`: Pauses the currently playing video.
3. `currentTime`: Represents the current playback time in seconds. It's both a getter and setter, meaning you can set this to a specific value to seek to that time in the video.

---

### Setting up the HTML:

Let's assume you have the following HTML structure for your video player:

```
``html
<div class="video-container">
 <video id="videoPlayer" width="640" height="360"
controls>
 <source src="path_to_your_video.mp4"
type="video/mp4">
 Your browser does not support the video tag.
</video>
<div class="controls">
 <button id="playBtn">Play</button>
 <button id="pauseBtn">Pause</button>
 <button id="stopBtn">Stop</button>
</div>
</div>
``
```

---

### Implementing Play, Pause & Stop functionalities:

#### 1. Play Video:

To play the video, we need to target our video element and call the `play()` method.

```
``javascript
```

```
const video = document.getElementById('videoPlayer');
document.getElementById('playBtn').addEventListener('click', function() {
 video.play();
});
```

## 2. Pause Video:

Pausing the video is just as simple. Target the video element and call the `pause()` method.

```
``javascript
document.getElementById('pauseBtn').addEventListener('click', function() {
 video.pause();
});
```

## 3. Stop Video:

Stopping the video involves two steps:

1. Pause the video.
2. Reset the video's current time to 0 (i.e., the beginning).

```
``javascript
document.getElementById('stopBtn').addEventListener('click', function() {
 video.pause();
 video.currentTime = 0;
});
```

---

Enhancing UX with Play/Pause Toggle:

Often, video players combine the play and pause functionality into one button. Let's create a toggle function for this:

1. Modify the HTML:

```
``html
<button id="togglePlayPauseBtn">Play</button>
``
```

2. JavaScript:

```
``javascript
const togglePlayPauseBtn =
document.getElementById('togglePlayPauseBtn');

togglePlayPauseBtn.addEventListener('click', function() {
 if (video.paused) {
 video.play();
 togglePlayPauseBtn.innerText = "Pause";
 } else {
 video.pause();
 togglePlayPauseBtn.innerText = "Play";
 }
});
``
```

---

Wrapping Up:

With the HTML5 Video API, implementing play, pause, and stop functionality becomes a walk in the park. The API offers a wide range of methods and properties that can help developers create robust and feature-rich video players.

In the next chapter, we'll dive into enhancing our video player's user experience by adding a progress bar and a

timestamp, allowing users to visually see the playback status and duration of the video.

Remember, practice makes perfect! Don't hesitate to play around with the code, try different functionalities, and make this video player truly your own. Happy coding!

## Video Progress Bar & Timestamp

Welcome to Chapter 19! In this chapter, we'll explore two critical elements of a custom video player – the progress bar and timestamp. Both of these enhance the user experience, allowing viewers to know the video's progress and control playback.

---

### 1. Understanding the Video API

Before diving into the coding part, it's essential to understand the HTML5 Video API a bit. The Video API offers properties like `currentTime` (current playback position, in seconds) and `duration` (total video duration, in seconds). These properties will be crucial for our progress bar and timestamp functionality.

---

### 2. HTML Structure

For our custom video player, let's consider you already have the following basic HTML structure from the previous chapters:

```
``html
<video id="video" width="750" controls>
 <source src="path_to_video.mp4" type="video/mp4">
</video>
<div id="video-controls">
```

```
<div id="progress-bar">
 <div id="progress"></div>
</div>
00:00 / 00:00
</div>
``
```

Here, `progress-bar` is the container, and `progress` is the fill that'll show video playback progress. The `timestamp` will display the current time and total video duration.

---

### 3. CSS for Progress Bar

Before adding functionality, ensure the progress bar looks right:

```
``css
progress-bar {
 width: 90%;
 height: 5px;
 background: e0e0e0;
 margin: 10px auto;
 position: relative;
}
progress {
 height: 5px;
 background: 007BFF;
 width: 0;
}
```

---

### 4. JavaScript Functionality

Now, let's dive into the fun part – the JavaScript!

### a. Progress Bar

We want the progress bar to fill up as the video plays:

```
``javascript
const video = document.getElementById('video');
const progressBar =
document.getElementById('progress-bar');
const progress = document.getElementById('progress');
video.addEventListener('timeupdate',
updateProgressBar);

function updateProgressBar() {
 const percentage = (video.currentTime /
video.duration) * 100;
 progress.style.width = `${percentage}%`;
}
``
```

### b. Timestamp

To update the timestamp:

```
``javascript
const timestamp =
document.getElementById('timestamp');
video.addEventListener('timeupdate',
updateTimestamp);

function updateTimestamp() {
 const minutesCurrent = Math.floor(video.currentTime
/ 60);
 let secondsCurrent = Math.floor(video.currentTime %
60);
 if (secondsCurrent < 10) {
 secondsCurrent = '0' + secondsCurrent;
 }
 timestamp.textContent = `${minutesCurrent} : ${secondsCurrent}`;
}
``
```

```
}

const minutesDuration = Math.floor(video.duration / 60);

let secondsDuration = Math.floor(video.duration % 60);

if (secondsDuration < 10) {

 secondsDuration = '0' + secondsDuration;

}

timestamp.textContent =
`${minutesCurrent}:${secondsCurrent} /
${minutesDuration}:${secondsDuration}`;

}

``
```

---

## 5. Seeking Functionality

Let's add the ability for users to click on the progress bar to seek to different parts of the video:

```
``javascript

progressBar.addEventListener('click', setVideoProgress);

function setVideoProgress(e) {

 const clickPosition = e.offsetX;

 const width = progressBar.offsetWidth;

 const clickPercentage = (clickPosition / width);

 video.currentTime = clickPercentage * video.duration;

}

``
```

---

## 6. Recap

By now, you have a functional progress bar that fills up as the video plays and a timestamp that updates in real-

time. The user can also click on the progress bar to seek the video, enhancing user control and experience.

## Section 5: Project 4 - Exchange Rate Calculator | Fetch & JSON Intro

### Project Intro

Welcome to Project 4, the Exchange Rate Calculator! In this project, we will embark on a journey to create a utility tool that can be widely used in daily life. Whether you're traveling, investing, or just curious about the world economy, an exchange rate calculator provides essential information about currency values in different countries.

---

#### Objective:

By the end of this project, you will have built an interactive web application that retrieves real-time exchange rates for various currencies and performs calculations to convert amounts between these currencies. This will be done using pure vanilla JavaScript, giving you a hands-on experience with the Fetch API and JSON data manipulation.

---

#### Project Overview:

Here's a sneak peek into what we will be developing:

1. User Interface: A sleek and intuitive design with dropdowns for selecting currencies and an input for the amount to convert.
2. Real-time Data: We'll fetch real-time exchange rates from a free API.

3. Conversion Logic: Convert input currency to the desired output currency and display the result.
  4. Error Handling: Implement proper error checking and handling to ensure a seamless user experience.
- 

#### Skills & Techniques to be Acquired:

- Fetch API: Understand how to make asynchronous requests to retrieve real-time data.
  - JSON Data Handling: Work with JSON responses to extract, manipulate, and use data in our application.
  - Dynamic DOM Manipulation: Populate dropdowns with available currencies, display real-time conversion rates, and more.
  - Error Handling: Create user-friendly error messages for issues like network errors or invalid data.
- 

#### Why This Project?

The Exchange Rate Calculator will not just be another project in your portfolio; it showcases your ability to build practical real-world applications. It emphasizes:

1. Utility: Creating something that people can use in their daily lives.
  2. Data Manipulation: Working with live data feeds and displaying them dynamically.
  3. Asynchronous Programming: Mastering Fetch API and asynchronous JS will open doors to many other projects in the future.
- 

#### Pre-requisites:

Before diving into the coding part, make sure you have a basic understanding of:

- HTML: For creating the basic structure of our application.

- CSS: To style our application and make it visually appealing.
  - JavaScript: Basics of variables, functions, and events.
- If you're a beginner or want a refresher, I'd recommend revisiting the chapters on HTML, CSS, and introductory JavaScript from this book or checking out my other courses on Udemy.
- 

### Conclusion:

With a clear goal in mind and excitement in our hearts, let's get started on building this project! The subsequent chapters will guide you step by step, from laying out the HTML structure, styling it with CSS, to bringing it to life with JavaScript. By the end, you'll have a fully functional Exchange Rate Calculator and a deeper understanding of web development techniques.

Are you ready to dive deep into the world of Fetch and JSON? Let's begin!

## Project HTML

In this chapter, we're going to lay the foundation for our Exchange Rate Calculator project. The HTML structure will form the skeleton upon which we will later add styling (CSS) and dynamic functionality (JavaScript). By the end of this chapter, you will have a clear understanding of the layout and elements that comprise our exchange rate calculator.

---

### Structure Overview

Our Exchange Rate Calculator will consist of:

- A title
- Two dropdown lists for selecting source and target currencies

- Two input fields for entering and displaying the conversion result
  - A button to perform the conversion
  - A section to display the current exchange rate between the selected currencies
- 

## The HTML Structure

```
“html
<!DOCTYPE html>
<html lang=“en”>
<head>
 <meta charset=“UTF-8”>
 <meta name=“viewport” content=“width=device-width,
initial-scale=1.0”>
 <title>Exchange Rate Calculator</title>
 <!-- Link to the CSS file will be added later -->
</head>
<body>
 <div class=“calculator-container”>
 <h1>Exchange Rate Calculator</h1>
 <div class=“input-group”>
 <label for=“from-currency”>From:</label>
 <select id=“from-currency”>
 <!-- Options will be populated using JavaScript
→
 </select>
 <input type=“number” id=“from-amount”
placeholder=“Enter amount”>
 </div>
 <div class=“input-group”>
```

```
<label for="to-currency">To:</label>
<select id="to-currency">
 <!-- Options will be populated using JavaScript
-->
</select>
<input type="number" id="to-amount"
placeholder="Converted amount" readonly>
</div>
<button id="convert-btn">Convert</button>
<div class="rate-display">
 <p>Exchange rate from to is: </p>
</div>
</div>
<!-- Link to the JavaScript file will be added later -->
</body>
</html>
"
```

---

## Breaking Down the Structure

### 1. HTML Boilerplate:

The beginning of our document sets up the standard structure, including the doctype declaration, `<head>` section, and `<body>` tag. We've also specified a `viewport` meta tag to ensure our calculator is mobile responsive.

### 2. Calculator Container:

We've wrapped our entire calculator in a `div` with the class `calculator-container`. This will help us style the calculator as a cohesive unit later on.

### 3. Title:

A simple heading (`<h1>`) announces the purpose of our application: “Exchange Rate Calculator”.

### 4. Input Groups:

We have two `div` elements with the class `input-group` for our source and target currencies. Inside each `div`, we have:

- A label
- A dropdown list (`<select>`) to choose a currency
- An input field (`<input>`) for amounts

The source currency input allows user entry, while the target currency input is read-only since it will display the calculated result.

### 5. Convert Button:

The “Convert” button (`<button>`) will trigger the currency conversion when clicked.

### 6. Rate Display:

Finally, we have a `div` with the class `rate-display` that will show the current exchange rate between the two selected currencies. We’ve used placeholder `<span>` elements to make it easier to insert dynamic data with JavaScript later on.

---

## Conclusion

By the end of this chapter, we’ve set up the essential structure for our Exchange Rate Calculator using pure HTML. We’ve ensured that our structure is clear and semantic, making it accessible and easy to style and script in the upcoming chapters. In the next chapters, we’ll look into styling this structure with CSS and then adding the dynamic functionalities with JavaScript.

## Project CSS

Welcome to Chapter 22, where we're going to focus on styling our Exchange Rate Calculator project. The design will be clean, modern, and intuitive. Given the importance of user experience in web applications, our goal is to create a user-friendly interface that's easy to navigate.

---

## 1. Setting Up the Basic Structure:

Before diving into the specifics, it's crucial to ensure you have your base CSS reset. This avoids any unwanted default styles applied by browsers.

```
``css
* {
 margin: 0;
 padding: 0;
 box-sizing: border-box;
 font-family: 'Arial', sans-serif;
}
``
```

---

## 2. The Main Container:

Our exchange rate calculator will be centered on the page. This main container will house all our components.

```
``css
.container {
 width: 90%;
 max-width: 600px;
 margin: 50px auto;
 background-color: f4f4f4;
 padding: 20px;
 border-radius: 5px;
```

```
 box-shadow: 0 3px 10px rgba(0, 0, 0, 0.2);
}
“
```

---

### 3. Styling the Heading:

Let's ensure our project heading stands out and provides a clear purpose to our users.

```
“css
.container h1 {
 text-align: center;
 margin-bottom: 20px;
 color: 333;
}
“
```

---

### 4. The Exchange Rate Input Fields:

The exchange rate calculator will primarily consist of two input fields – one for the source currency and one for the target currency.

```
“css
.input-group {
 display: flex;
 justify-content: space-between;
 margin-bottom: 15px;
}
.input-group label {
 flex: 1;
 padding: 10px;
 background: ddd;
```

```
border: 1px solid bbb;
border-right: none;
border-radius: 5px 0 0 5px;
}
.input-group input {
 flex: 2;
 padding: 10px;
 border: 1px solid bbb;
 border-radius: 0 5px 5px 0;
}
“
```

---

## 5. Currency Dropdowns:

We'll have two dropdowns for users to select currencies they want to exchange between.

```
“css
.select-group {
 margin-bottom: 20px;
}
.select-group select {
 width: 48%;
 padding: 10px;
 border: 1px solid bbb;
 border-radius: 5px;
}
“
```

---

## 6. The Result Display:

Once the user inputs an amount and selects the relevant currencies, the converted value will be displayed below.

```
``css
.result-display {
 background-color: e6e6e6;
 padding: 10px;
 border-radius: 5px;
 text-align: center;
 margin-top: 20px;
}
.result-display p {
 font-size: 1.5em;
 color: 333;
}
``
```

---

## 7. Button Styling:

Our calculate button will trigger the exchange rate calculation.

```
``css
.calculate-btn {
 width: 100%;
 padding: 10px;
 background-color: 0099cc;
 border: none;
 border-radius: 5px;
 color: fff;
 font-size: 1.2em;
 cursor: pointer;
}
```

```
 transition: background-color 0.3s ease;
 }
.calculate-btn:hover {
 background-color: 0077aa;
}
``
```

---

## 8. Responsive Design:

We want our calculator to look good on both desktops and mobile devices.

```
``css
```

```
@media (max-width: 600px) {
 .input-group, .select-group {
 flex-direction: column;
 }
 .input-group label, .input-group input, .select-group select {
 width: 100%;
 margin-bottom: 10px;
 }
}
```

---

## Conclusion:

With this CSS setup, our Exchange Rate Calculator should now have a sleek, modern look. This ensures that not only does our application work efficiently, but it also provides a pleasant user experience. Remember, CSS is all about experimentation and creativity, so don't hesitate to adjust the styles to suit your personal design preferences.

In the next chapter, we'll dive into understanding JSON and how to use the Fetch API to retrieve current exchange rates. Stay tuned!

## A Look at JSON & Fetch

In this chapter, we'll dive deep into two crucial concepts in modern web development: JSON and the Fetch API. As we proceed with our Exchange Rate Calculator, we will rely on these tools to fetch and handle data from external sources. By understanding their foundations, you'll be better equipped to create dynamic, data-driven web applications.

---

### What is JSON?

JSON stands for JavaScript Object Notation. It's a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is a text format, independent of any language, which makes it an ideal data format for data exchange between a client and server, or between different parts of an application.

A JSON object looks like this:

```
“json
{
 “name”: “John”,
 “age”: 30,
 “city”: “New York”
}
“
```

### Key Features of JSON:

1. Readability: JSON structures are straightforward, making it easy for humans to read and write.

2. Universality: Most programming languages, including JavaScript, have built-in support for JSON.

3. Flexibility: JSON can represent a wide variety of data structures, including objects, arrays, and primitive data types.

---

## Introduction to Fetch API

The Fetch API provides an interface for fetching resources, including across the network. It gives a more modern way to make web requests compared to the older `XMLHttpRequest`. With `fetch()`, we can make requests and handle responses more flexibly and with less boilerplate code.

Basic Syntax of Fetch:

```
``javascript
fetch(url)
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => console.error('Error:', error));
``
```

---

## Combining JSON & Fetch

When we use the Fetch API to retrieve data from a server, the response is often in JSON format. Here's a step-by-step breakdown of how to use them together:

1. Request Data: Use the `fetch()` function to make a request to an API or server.

2. Handle Response: Once the request is made, the server will send back a response. This response is not directly in JSON format, but a `Response` object which contains the data in a method like `json()`.

3. Parse JSON: Using the `json()` method, we can extract the JSON data from the `Response` object.

Example:

```
``javascript
fetch('https://api.example.com/data')
.then(response => response.json())
.then(data => {
 // Handle the JSON data here
 console.log(data);
})
.catch(error => {
 console.error('Error fetching data:', error);
});
``
```

---

## Practical Application: Fetching Exchange Rates

For our Exchange Rate Calculator project, we'll be using an API that provides currency exchange rates in JSON format. Using the Fetch API, we can retrieve this data, parse the JSON, and then manipulate it within our application to provide users with real-time currency conversions.

Here's a hypothetical example:

```
``javascript
fetch('https://api.currencyexchange.com/rates')
.then(response => {
 if (!response.ok) {
 throw new Error('Network response was not ok');
 }
 return response.json();
})
.then(data => {
```

```
// Use the data (exchange rates) to update our
application's UI

 updateExchangeRates(data);
})

.catch(error => {
 console.error('There was a problem with the fetch
operation:', error.message);
});

``
```

In the above example, after fetching the data, we check if the response was successful using the `response.ok` property. If there's an issue (e.g., the server returns a 404 or 500 status), we handle it appropriately.

---

## Conclusion

Understanding JSON and the Fetch API is essential for modern web development. These tools allow us to interact with servers, APIs, and other external data sources seamlessly. As we build our Exchange Rate Calculator, we'll see these concepts in action, retrieving real-world data and presenting it to our users. In the next chapters, we'll integrate this knowledge into our project, setting the foundation for dynamic, data-driven web applications.

---

## Further Reading & Resources:

- MDN Web Docs: [Using Fetch]  
([https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch))
- MDN Web Docs: [JSON]  
([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/JSON](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON))

### Exercise:

1. Visit an online JSON editor, such as [jsoneditoronline.org](<https://jsoneditoronline.org/>). Create a sample JSON structure and practice modifying it.
2. Experiment with the Fetch API. Try fetching data from a public API like [jsonplaceholder.typicode.com](<https://jsonplaceholder.typicode.com/>). Analyze the JSON response you get.

---

With this understanding of JSON and Fetch, you're now ready to dive deeper into the Exchange Rate Calculator project and apply what you've learned to build a functional, interactive application.

## Fetch Rates & Update DOM

In the preceding chapters, we laid the groundwork for our Exchange Rate Calculator project, structuring our HTML and beautifying it with CSS. Now, we'll dive into the heart of our application: fetching real-time exchange rates and dynamically updating our webpage (or the Document Object Model, abbreviated as DOM).

---

### 1. Introduction to Fetching Data

Fetching data from external sources is a common requirement in modern web applications. JavaScript provides a built-in method called `fetch()` to make HTTP requests, which simplifies the process of retrieving and sending data to an external source.

---

### 2. Fetch API

The Fetch API provides a simple interface for making network requests and handling responses. The primary method for this is `fetch()`.

## Basic Syntax:

```
``javascript
fetch(url)
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => console.error("There was an error!",
error));
``
```

- `url`: The URL you want to fetch.
- The `fetch()` method returns a promise that resolves into the Response object representing the response to the request.
- The `response.json()` method reads the response body and returns it as a JSON object.

---

## 3. Setting Up the Exchange Rate API

For this project, we'll use the `exchangerate-api` (or any similar service). It's a free service that provides currency conversion rates.

Firstly, sign up and get your API key.

## 4. Making Our Fetch Request

With our API key at the ready, let's fetch the rates:

```
``javascript
const currencyOne =
document.getElementById('currency-one');

const currencyTwo =
document.getElementById('currency-two');

const rateElement = document.getElementById('rate');

let apiURL = `https://api.exchangerate-
api.com/v4/latest/${currencyOne.value}`;

fetch(apiURL)
```

```
.then(response => response.json())
.then(data => {
 const rate = data.rates[currencyTwo.value];
 rateElement.innerText = `1 ${currencyOne.value} = ${rate} ${currencyTwo.value}`;
})
.catch(error => console.error("Error fetching data:", error));
``
```

Here, we're fetching the latest exchange rate for the currency selected in `currencyOne` and updating the DOM with the rate for `currencyTwo`.

## 5. Updating the DOM

After fetching the data, our main task is to dynamically update our webpage. We've already displayed the rate in the above code. But to make our application interactive, let's also convert the entered amount:

```
``javascript
const amountOne = document.getElementById('amount-one');
const amountTwo = document.getElementById('amount-two');
fetch(apiURL)
 .then(response => response.json())
 .then(data => {
 const rate = data.rates[currencyTwo.value];
 rateElement.innerText = `1 ${currencyOne.value} = ${rate} ${currencyTwo.value}`;
 // Convert the amount
 const amount = amountOne.value * rate;
 })
``
```

```
amountTwo.value = amount.toFixed(2);
});
``
```

We've added code to convert `amountOne` using the fetched rate and update `amountTwo` accordingly.

## 6. Event Listeners

To make our application responsive to user actions, let's add event listeners:

```
“javascript
currencyOne.addEventListener('change', calculate);
currencyTwo.addEventListener('change', calculate);
amountOne.addEventListener('input', calculate);
amountTwo.addEventListener('input', calculate);
”
```

The `calculate` function will call our fetching logic and update the DOM. Now, every time a user changes the currency type or adjusts the amount, our application will dynamically fetch the new rate and update the DOM.

## 7. Error Handling

The world of APIs isn't always perfect. There can be downtimes, or we might exceed our API request limits. Let's add some basic error handling:

```
``javascript
fetch(apiURL)
 .then(response => {
 if(!response.ok) {
 throw new Error("Network response was not ok");
 }
 return response.json();
 })
 .catch(error => {
 console.error("There has been a problem with your fetch operation: " + error.message);
 });
``
```

```
})
.then(data => {
 //... rest of the logic
})
.catch(error => {
 console.error("There was a problem with the fetch
operation:", error.message);
});
```

```

8. Conclusion

With this, our Exchange Rate Calculator's core functionality is complete! You've successfully incorporated an external API into your project and made your application dynamic and interactive.

In subsequent projects, you'll explore more about asynchronous JavaScript and how to further interact with APIs and the DOM. Remember, practice is key. The more projects you build, the more proficient you'll become.

Note: Always ensure you handle data from external sources with care. Ensure that the data you fetch is from a trusted source, and always validate and sanitize your data before using it.

Section 6:

Project 5 - DOM Array Methods | forEach, map, filter, sort, reduce

Project Intro

Welcome to Project 5: DOM Array Methods Exploration! If you've ever been intrigued by the powerful capabilities JavaScript offers when working with arrays, you're in the right place. This project will provide an insightful journey into some of the most widely-used and fundamental array methods in JavaScript.

Why DOM Array Methods?

Arrays are among the most basic yet crucial data structures in JavaScript. Whether you're creating a simple list of items, handling data from an API, or building more complex applications, you'll find yourself working with arrays. JavaScript offers a rich collection of built-in methods to help you manage and manipulate these arrays efficiently.

But why combine DOM with array methods? The Document Object Model (DOM) represents the structure of your web pages. By combining our knowledge of the DOM with array methods, we can create dynamic and interactive web applications. This project will demonstrate the sheer power of this combination.

What Will We Build?

We will develop an application that fetches random user data from an API, presents it on a webpage, and allows users to interact with the data using various functionalities. Here's a brief overview:

- Generate Random Users: We'll fetch user data and showcase it on our website.
- Double Money: With the click of a button, we'll demonstrate how to manipulate numerical data in our array, doubling the money of our users.

- Sort by Richest: Another interactive feature to sort our users based on their wealth.
 - Show Millionaires: A filter functionality to display only those users who have a wealth exceeding a million.
 - Calculate Total Wealth: We'll sum up the wealth of all our users and present it in a neat format.
-

Learning Outcomes

By the end of this project, you will:

1. Deepen Your Understanding of Array Methods: We'll delve deep into methods like `forEach`, `map`, `filter`, `sort`, and `reduce`. You'll understand not just how, but also why and when to use them.
 2. Enhance DOM Manipulation Skills: You'll get a lot of practice dynamically adding, removing, and altering elements on a webpage.
 3. Strengthen Async Operations: We'll be fetching data from an API, giving you a practical scenario to understand asynchronous operations in JavaScript.
 4. Boost Problem-Solving Abilities: Through hands-on tasks, you'll improve your logical thinking and problem-solving skills.
-

Prerequisites

To make the most of this project:

- A basic understanding of HTML, CSS, and JavaScript is required. If you're a beginner, I'd recommend revisiting the earlier chapters on these topics.
- An understanding of how to fetch data from APIs will be beneficial. If you're unfamiliar, don't worry! We'll cover the basics in this project.
- Patience and a willingness to experiment. Sometimes, the best way to understand a concept is to play around with it, make mistakes, and learn from them.

A Word Before We Begin

Every developer, novice or expert, goes through a journey of discovery and learning. This project is designed as a stepping stone in your path to mastering web development. While the focus is on array methods and DOM manipulation, remember that the broader goal is to nurture a mindset of exploration and continuous learning.

Dive in with enthusiasm, keep an open mind, and most importantly, enjoy the process! Let's embark on this exciting journey together.

Project UI

In the world of web development, the user interface (UI) is the space where interactions between humans and machines occur. The goal of this interaction is to allow effective operation and control of the machine from the human end, while the machine simultaneously provides feedback that aids the operators' decision-making process. In "Project 5: DOM Array Methods", our main objective is to showcase an array of different JavaScript methods, and to effectively do so, we need a fitting UI.

The purpose of this project is to build a dynamic user dashboard that will list out random users, their wealth, and provide us with various functionalities to manipulate this data. The user will be able to double the money for users, sort by the richest, filter to show only millionaires, and calculate total wealth. Let's get started with crafting this UI!

Setting up the Structure:

HTML Structure:

1. Main Container - This container will encapsulate all the UI elements related to our project.

2. Header - A simple header to display the name of our application/project.
3. Users List - A structured list where we'll display each user and their wealth.
4. Action Buttons - This is a set of buttons that will enable the different functionalities we want to showcase.

```
``html
```

```
<div class="main-container">  
  <header>  
    <h1>DOM Array Methods Dashboard</h1>  
  </header>  
  <ul class="users-list"></ul>  
  <div class="actions">  
    <button id="double">Double Money</button>  
    <button id="show-millionaires">Show  
      Millionaires</button>  
    <button id="sort">Sort By Richest</button>  
    <button id="calculate-wealth">Calculate Total  
      Wealth</button>  
  </div>  
</div>
```

``

Styling the UI:

CSS Structure:

1. Main Container - This will be centered in the middle of the screen with some padding for aesthetics.
2. Header - We'll give it a font-size enhancement, center-align the text, and add some margins for spacing.
3. Users List - Here, we will style it so each user and their wealth are displayed in neat rows with alternate

light and dark backgrounds for better clarity.

4. Action Buttons - These will be styled to be easily recognizable, with hover effects to show interactivity.

``css

```
.main-container {  
    width: 80%;  
    margin: auto;  
    padding: 2rem;  
    border-radius: 10px;  
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);  
}  
  
header {  
    text-align: center;  
    margin-bottom: 2rem;  
}  
  
.users-list {  
    list-style-type: none;  
    padding: 0;  
}  
  
.users-list li:nth-child(odd) {  
    background-color: f4f4f4;  
}  
  
.users-list li {  
    padding: 1rem;  
    display: flex;  
    justify-content: space-between;  
    align-items: center;  
}
```

```
.actions button {  
    padding: 0.5rem 1rem;  
    margin: 0.5rem;  
    border: none;  
    border-radius: 5px;  
    cursor: pointer;  
    transition: background-color 0.3s ease;  
}  
.actions button:hover {  
    background-color: e0e0e0;  
}  
"
```

Conclusion:

The UI plays a crucial role in ensuring our project's array methods are both understood and appreciated by the end-user. By setting up a simple yet effective interface, we pave the way for the logical JavaScript functions that will populate and manipulate the data on this interface.

In the upcoming chapters, we'll delve deeper into fetching random user data, and employing the array methods to provide the functionalities we've hinted at with our action buttons. Stay tuned and keep coding!

Generate Random Users - Fetch w/ Async/Await

Generating random users for a project can be essential, especially when building applications that require user data but you don't want to use actual personal data. In this chapter, we will be fetching random user data using the Random User Generator API. The Fetch API

combined with Async/Await makes the process of obtaining this data smooth and efficient.

Setting Up The API

Before we dive into the code, you need to be familiar with the API we will be using. The Random User Generator is a free, open-source API that provides random user data in JSON format. The data can include things like name, picture, location, email, and more.

API Endpoint: `https://randomuser.me/api/`

Getting Started with Fetch and Async/Await

Before we work with Async/Await, let's understand the basics of the Fetch API.

The Fetch API provides an interface for fetching resources. In simpler words, it lets you communicate with other websites and servers, which is essential when we want to retrieve or send data.

A basic fetch request looks like this:

```
``javascript
fetch('https://randomuser.me/api/')
  .then(response => response.json())
  .then(data => console.log(data));
``
```

However, we'll be utilizing `async/await` to make our code cleaner and easier to understand. With `async/await`, our code becomes:

```
``javascript
async function fetchUsers() {
  let response = await
    fetch('https://randomuser.me/api/');
  let data = await response.json();
``
```

```
        console.log(data);
    }
fetchUsers();
```

```

---

## Generating Random Users

To fetch multiple users, we need to specify the number of users we want by adding a parameter to the API endpoint.

For instance, to fetch 5 users:

```
`https://randomuser.me/api/?results=5`
```

Let's integrate this into our function:

```
``javascript
async function fetchMultipleUsers(num) {
 let response = await
fetch(`https://randomuser.me/api/?results=${num}`);
 let data = await response.json();
 displayUsers(data.results); // We will create this
function next
}
fetchMultipleUsers(5);
``
```

---

## Displaying the Users

Now that we've fetched the users, we want to display them in our HTML. For this example, we'll create a simple list of names.

```
``javascript
function displayUsers(users) {
 const usersContainer =
document.getElementById('usersContainer');
```

```
users.forEach(user => {
 const userElement =
document.createElement('div');
 userElement.innerHTML = `
 <h2>${user.name.title} ${user.name.first}
${user.name.last}</h2>

 <p>Email: ${user.email}</p>
 <p>Location: ${user.location.city},
${user.location.state}</p>
 `;
 usersContainer.appendChild(userElement);
});

}

``
```

Ensure you have an element in your HTML with the id `usersContainer`:

```
``html
<div id="usersContainer"></div>
``
```

---

## Handling Errors

When working with APIs, it's crucial to handle errors gracefully. Network problems, API downtimes, or request limits can cause issues. With `async/await`, we can use simple try/catch blocks:

```
``javascript
async function fetchMultipleUsers(num) {
 try {
```

```
let response = await
fetch(`https://randomuser.me/api/?results=${num}`);
if (!response.ok) {
 throw new Error(`HTTP error! Status:
${response.status}`);
}
let data = await response.json();
displayUsers(data.results);
} catch (error) {
 console.error('Fetch error: ' + error.message);
}
```

```

Conclusion

In this chapter, we learned how to generate random user data using the Random User Generator API, integrated with the Fetch API and Async/Await. This technique can be instrumental when you want to create mock data for your applications, especially during the development phase.

Remember, always refer to the API's documentation if you wish to explore more features or if you run into any issues. APIs are powerful tools, and mastering them can elevate your projects significantly.

Output Users - `forEach()` & DOM Methods

Welcome to Chapter 28! In this chapter, we will take a deep dive into how we can output a list of users to our web page using the `'forEach()'` method and DOM

manipulation techniques. We will be working on the DOM Array Methods project, and our goal is to fetch a list of random users and display them in a user-friendly format.

Prerequisites:

Before jumping into this chapter, make sure you have:

1. Basic understanding of JavaScript and its ES6 syntax.
 2. Familiarity with the Document Object Model (DOM).
 3. Completed Chapter 27 where we generated a list of random users using Fetch with Async/Await.
-

Step 1: Setting Up Our HTML Structure

We need a section in our HTML where we will output our users. Let's create a simple `div` to hold our users:

```
``html
<div id="users-output"></div>
``
```

Step 2: Fetching Users

Assuming you've fetched the users in the previous chapter, you should have an array of user objects. For simplicity's sake, let's say our array looks something like this:

```
``javascript
const users = [
  {id: 1, name: 'John Doe', age: 32},
  {id: 2, name: 'Jane Smith', age: 28},
  // ... more users
];
``
```

Step 3: Outputting Users with `forEach()` & DOM Methods

Now, let's use the `forEach()` method to loop through each user and display them:

```
“javascript
const outputDiv = document.getElementById('users-
output');

users.forEach(user => {
    // Create a new div for each user
    const userDiv = document.createElement('div');
    userDiv.className = 'user';
    // Create an innerHTML template for the user
    userDiv.innerHTML =
        <h3>${user.name}</h3>
        <p>Age: ${user.age}</p>
    ;
    // Append the user div to the main output div
    outputDiv.appendChild(userDiv);
});
```

In the code above:

1. We first get a reference to our main output div.
2. Using `forEach()`, we loop through each user.
3. For each user, we create a new `div`.
4. We then set the `innerHTML` of that div to a template which displays the user's name and age.
5. Finally, we append this user div to the main output div.

Step 4: Styling Our Users

While this is mainly a JavaScript-focused chapter, adding a bit of CSS can help in presenting our users more attractively:

```
``css
.user {
    border: 1px solid ccc;
    padding: 10px;
    margin: 10px 0;
    border-radius: 5px;
}
``
```

This will give each user a nice box with some spacing between each one.

Summary:

In this chapter, we learned how to use the `forEach()` method in conjunction with DOM methods to output users to our web page. This is a fundamental pattern in web development – fetching data and then displaying it to the user.

In the next chapter, we will enhance our project by adding functionalities that allow users to double their money using the `map()` method. Stay tuned!

Note: Remember that in a real-world scenario, the user data would be fetched from an API or a database. The hard-coded `users` array is just for illustration purposes in this example. Also, always ensure you handle any errors that might occur during the fetch operation or any other potential issues to enhance the user experience.

Double Money - map()

In this chapter, we will explore one of the highly useful array methods in JavaScript - the `map()` function. It's a part of the DOM array methods we're delving into in this project. As the chapter title suggests, we'll be applying `map()` to double the money values in our list. This practical exercise will give you a hands-on experience of how the `map()` function operates and how you can leverage it in various web projects.

What is map()?

The `map()` method is a higher-order function that creates a new array populated with the results of calling a provided function on every element in the calling array. Unlike `forEach()`, which doesn't return anything, `map()` returns a new array without mutating the original array.

Basic Syntax:

```
``javascript
const newArray = array.map(function(currentValue,
index, arr), thisValue);
``
```

- `currentValue`: The current element being processed in the array.
 - `index` (optional): The index of the current element being processed.
 - `arr` (optional): The array on which map was called.
 - `thisValue` (optional): Object to use as `this` when executing the function.
-

Practical Application: Doubling Money

Imagine you have a list of users, each with an account balance. Perhaps you're running a special promotion where you're offering to double the balance of every user. Let's see how you can use the `map()` function to achieve this.

Step 1: Creating our Original Array

First, let's create our initial array of users with their current balances:

```
``javascript
const users = [
  { name: 'Alice', balance: 100 },
  { name: 'Bob', balance: 200 },
  { name: 'Charlie', balance: 150 },
  { name: 'David', balance: 80 }
];
``
```

Step 2: Using map() to Double the Money

To double the money for each user, we'll utilize the `map()` function. We'll target the `balance` property and multiply it by 2:

```
``javascript
const doubledBalances = users.map(user => {
  return {
    ...user,
    balance: user.balance * 2
  };
});
``
```

In the code snippet above, we're returning a new object for each user. We spread out the original user properties using the spread operator (`...user`) and then overwrite the balance property by multiplying its current value by 2.

Step 3: Display the Results

Now, let's display our results to see the original balances and the doubled balances:

```
``javascript
console.log("Original Balances:", users);
console.log("Doubled Balances:", doubledBalances);
``
```

Why use map()?

1. Immutability: One of the biggest advantages of `map()` is that it doesn't mutate the original array. This is in line with functional programming principles, ensuring data remains unchanged, which can help prevent unintended side-effects in your code.
2. Chainability: Since `map()` returns a new array, you can chain other array methods like `sort()`, `filter()`, etc., making your code more concise and readable.
3. Flexibility: The provided function can be as simple or complex as needed, allowing for a wide range of transformations on array elements.

Conclusion

The `map()` function is a powerful tool in your JavaScript toolkit. It provides an easy, efficient, and functional way to transform data in arrays. In this chapter, we leveraged `map()` to double the money for each user in our array. However, its applications are diverse, and as you progress through different web projects, you'll find countless scenarios where `map()` will come in handy.

Sort By Richest - sort()

In our journey through the realm of web development, we've tackled various DOM Array methods, each with its unique powers. In this chapter, we'll delve into the `sort()` method. It might seem like a simple tool to arrange items, but with JavaScript, it becomes a versatile utility that can be tailored for a wide range of

use-cases. Our mission? To sort a list of users by their wealth, showcasing those who have the most at the top.

Understanding the `sort()` Method

At its core, the `sort()` method is used to arrange the elements of an array based on some comparison criteria. By default, it sorts elements as strings. So, `[10, 2, 22].sort()` would return `[10, 2, 22]`. This might not be what you'd expect, but it's how JavaScript works by default.

But fear not, for the `sort()` method allows us to define our own comparison function, granting us the power to customize how we want our elements to be sorted.

Syntax of `sort()`

```
“`javascript
array.sort([compareFunction])
“`
```

The `compareFunction` is optional, but crucial for our task. This function should return:

- A negative value if `a` should be sorted before `b`
 - A positive value if `a` should be sorted after `b`
 - Zero if `a` and `b` are equal
-

Sorting Users by Wealth

Let's imagine we have an array of user objects, each with a `name` and a `wealth` property:

```
“`javascript
const users = [
  { name: ‘John’, wealth: 2300 },
  { name: ‘Sarah’, wealth: 5800 },
  { name: ‘Mike’, wealth: 4500 },
```

```
];
```

```
"
```

To sort these users by their wealth in descending order (richest first), we'd utilize the `sort()` method as follows:

```
"javascript
```

```
users.sort((a, b) => b.wealth - a.wealth);
```

```
"
```

Here, `(a, b)` are two consecutive elements in the `users` array. The subtraction results in a positive or negative value, helping `sort()` decide the order.

Visualizing the Sorting Mechanism

Consider two users:

- User A with a wealth of 5000
- User B with a wealth of 3000

When sorting in descending order: `b.wealth - a.wealth`:

- For A and B: $3000 - 5000 = -2000$ (a negative value, so A comes before B)

Now you can see how the users get sorted with the richest at the top!

Displaying Sorted Users on the DOM

Once sorted, you might want to display the users in the DOM. Using the techniques we learned from earlier chapters, you can create and append elements to the DOM.

```
"javascript
```

```
const userList = document.querySelector('.users-list');
```

```
users.forEach(user => {
```

```
    const userElement = document.createElement('div');
```

```
    userElement.className = 'user';
```

```
userElement.innerHTML = `<h3>${user.name}</h3><p>${user.wealth.toLocaleString()}</p>`;
userList.appendChild(userElement);
});``
```

In the code above, we're iterating through each user, creating a new DOM element for them, and appending it to a `users-list` container. The `toLocaleString()` method is a neat trick to format numbers with commas (or appropriate locale symbols).

Conclusion

The `sort()` method, though appearing simple, is an essential tool in your JavaScript toolbox. With it, you've successfully sorted users by their wealth, enriching the functionality of your web application. This method's true strength lies in its flexibility—by defining our own comparison functions, we can sort arrays in numerous, customized ways.

Remember, web development isn't just about knowing the tools; it's about understanding them and innovating with how you use them. As you move forward, think about other exciting ways you can utilize `sort()` in your projects!

Practice Exercise

Try sorting the users in ascending order of their wealth. How would you modify the compare function to achieve this?

In the next chapter, we'll dive into the `filter()` method, a powerful tool to refine our lists based on specific criteria.

Stay tuned!

Show Millionaires - filter()

Welcome to Chapter 31, where we dive deep into one of the most commonly used array methods in JavaScript: the `filter()` method. Given our project's theme on DOM Array Methods, this chapter will specifically focus on using `filter()` to display only the millionaires from a list of users. We'll learn how to harness the power of `filter()` to create a subset of our original data array and display it dynamically on our webpage.

What is the filter() method?

The `filter()` method creates a new array with all the elements that pass the test implemented by the provided function. In simpler terms, it filters out elements based on a condition you set.

Syntax:

```
``javascript
```

```
let newArray = arr.filter(callback(element[, index[, array]])  
[, thisArg])
```

```
``
```

Where:

- `callback`: Function is a predicate, which returns a Boolean value. It tests each element and returns `true` to keep the element, `false` otherwise.
- `element`: Current element being processed in the array.
- `index` (optional): Index of the current element in the array.
- `array` (optional): The array `filter()` was called upon.

- `thisArg` (optional): Object to use as `this` when executing the callback.

Implementing the filter() method to show millionaires

Let's take a scenario where we have an array of user objects, each having properties like `name`, `age`, and `wealth`. Our objective is to filter out and display only those users who have wealth above 1 million.

Step 1: Sample Data Array

For the sake of this example, consider the following sample user data:

```
``javascript
const users = [
  { name: 'John Doe', age: 32, wealth: 500000 },
  { name: 'Jane Smith', age: 29, wealth: 1500000 },
  { name: 'Chris Johnson', age: 45, wealth: 2200000 },
  { name: 'Anna Brown', age: 28, wealth: 750000 },
  { name: 'Tom Davis', age: 34, wealth: 9800000 }
];
``
```

Step 2: Using filter() to get millionaires

The logic is simple: We need to test if the `wealth` property of each user is greater than 1 million.

```
``javascript
const millionaires = users.filter(user => user.wealth >
1000000);
console.log(millionaires);
``
```

The above code will give us an array of users who have wealth greater than 1 million.

Step 3: Displaying Millionaires on the DOM

We will create a function to dynamically display the list of millionaires on our webpage:

```
``javascript
function displayMillionaires(millionaires) {
    const millionaireContainer =
document.getElementById('millionaire-list');

    // Clear out the previous list
    millionaireContainer.innerHTML = "";

    // Loop through each millionaire and add to the DOM
    millionaires.forEach(millionaire => {
        const millionaireElement =
document.createElement('div');
        millionaireElement.className = 'millionaire';
        millionaireElement.innerHTML = `
            <strong>${millionaire.name}</strong> - Wealth:
            $$ ${millionaire.wealth.toLocaleString()}`;
        ;
        millionaireContainer.appendChild(millionaireElement);
    });
}

// Call the display function
displayMillionaires(millionaires);
``
```

Remember to have an element with the id `millionaire-list` in your HTML for this to work.

Conclusion

The `filter()` method is powerful, allowing us to extract subsets of data from our main dataset based on specific criteria. In this chapter, we successfully isolated and

displayed our millionaire users. As you progress through your JavaScript journey, you'll find countless scenarios where `filter()` becomes indispensable.

Calculate Wealth - reduce()

Welcome to another exciting chapter in our DOM Array Methods project. By now, we've covered a variety of array methods like `forEach()`, `map()`, `filter()`, and `sort()`. In this chapter, we'll focus on another powerful array method: `reduce()`. Our aim? To calculate the total wealth of a list of users.

Introduction to `reduce()`

The `reduce()` method applies a function to an accumulator and each element in an array (from left to right) to reduce it to a single value. The accumulator accumulates the callback's return values. If no initialValue is provided, the first element in the array will be used as the accumulator, and the callback will start from the second element.

Syntax:

```
``javascript
array.reduce(callback(accumulator, currentValue[, index[, array]][], initialValue])
``
```

Key Points:

- **accumulator:** This accumulates the callback's return values.
- **currentValue:** The current element being processed in the array.
- **index (optional):** The index of the `currentValue`.
- **array (optional):** The array `reduce()` was called upon.

- initialValue (optional): Value to use as the first argument to the first call of the callback.

Scenario: Calculating Total Wealth

Imagine you have a list of users, each with an `id`, `name`, and `wealth`:

```
``javascript
const users = [
    { id: 1, name: "John", wealth: 50000 },
    { id: 2, name: "Jane", wealth: 150000 },
    { id: 3, name: "Doe", wealth: 30000 },
    { id: 4, name: "Smith", wealth: 120000 }
];
``
```

We want to calculate the total wealth of all users.

Implementing with `reduce()`

Let's see how we can achieve this using the `reduce()` method:

```
``javascript
const totalWealth = users.reduce((acc, user) => acc +
    user.wealth, 0);
console.log(totalWealth); // Outputs: 350000
``
```

In the above example:

1. We provide an initial value of 0 for the accumulator `acc`.
2. For each `user` in the `users` array, we add the user's `wealth` to the accumulator.
3. The final value of `acc` (i.e., 350000 in this case) represents the total wealth.

Breaking it Down

- The `reduce()` function starts with the accumulator (`acc`) value of 0 (our initial value).
- For the first user (John), `acc` is 0 and `user.wealth` is 50000. So, `acc` becomes 50000.
- For the next user (Jane), `acc` is 50000 and `user.wealth` is 150000. So, `acc` becomes 200000.
- This process continues for all users.
- At the end of the iteration, `acc` holds the total wealth of all users.

Wrapping Up

The `reduce()` method offers a concise way to calculate aggregate values from arrays. In this chapter, we used it to calculate the total wealth from an array of users. But the `reduce()` function's applications don't end here. You can use it in numerous scenarios, like finding the maximum value in an array, counting occurrences of items, and more.

Section 7: Project 6 - Menu Slider & Modal | DOM & CSS

Project Intro

Welcome to the sixth project in our series, where we will be delving into the creation of a Menu Slider & Modal using pure HTML, CSS, and JavaScript. No frameworks, no libraries, just vanilla web technologies. As always, we're here to guide you every step of the way, ensuring you understand both the "how" and the "why" behind each part of the code.

1. Project Overview

In this project, we'll create an interactive navigation menu that slides in from the side (often referred to as a "hamburger menu"). This is a popular feature on many modern websites, providing a clean user interface, especially for mobile users.

But that's not all! To complement our menu slider, we'll also create a modal. A modal is a dialog box or pop-up window that is displayed on top of the current page, often used for notifications, gathering user input, or displaying additional information without leaving the page.

2. Core Concepts Covered

- * DOM Manipulation: This project will allow us to practice our skills in manipulating the Document Object Model (DOM), giving us the ability to interact with and modify web page elements dynamically.
- * CSS Transitions and Animations: We'll delve deeper into the world of CSS to make our menu slide effect and modal appearance smooth and aesthetically pleasing.
- * Event Listeners: These are crucial when creating interactivity. We'll use event listeners to detect when the menu button is clicked, when modal triggers are engaged, and when the user wants to close the modal.

3. What You Will Achieve

By the end of this project, you will have:

- * A slide-in menu that activates on a button click, providing a smooth transition effect.
- * A modal that can be triggered to open and will have a close button to hide it.
- * A better understanding of the DOM, and how to interact with it using vanilla JavaScript.

- * Enhanced your CSS skills, particularly in creating animations and transitions.
 - * More confidence in building interactive web components from scratch!
-

4. Pre-requisites

While this project is designed to be accessible for those with a basic understanding of HTML, CSS, and JavaScript, it will be beneficial if you've gone through the prior projects in this course. We'll be building upon foundational concepts introduced in earlier chapters.

5. A Peek at the End Goal

Imagine a sleek webpage. On the top left corner, there's a hamburger icon (three stacked lines). When clicked, a menu slides in from the left, revealing links to different sections of the website. This sliding effect is smooth, making the user experience feel polished and professional.

Now, within our main content, there's a button labeled "Learn More." Upon clicking it, the background dims, and a centered modal window appears, containing more details or perhaps a sign-up form. Clicking a close icon or anywhere outside the modal will hide it again, returning the user to the main content.

Sounds exciting, right? Well, that's what we are about to build! So, buckle up, and let's get started on this fantastic journey to add more interactive elements to your web development toolkit. On to the next chapter where we begin with our project's HTML structure!

Project HTML

Welcome to the HTML section of our sixth project: the Menu Slider & Modal! This is where we lay down the foundation of our project. Remember, a good structure in

your HTML is crucial to making your CSS and JavaScript integration easier and more efficient.

Introduction

HTML (HyperText Markup Language) is the backbone of any web page. It provides the structure, while CSS styles it and JavaScript brings it to life. In this chapter, we'll focus on creating a semantic and accessible structure for our Menu Slider and Modal.

Setting up the Basic HTML Structure

To begin with, we will set up a basic HTML5 template.

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Menu Slider & Modal</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
</body>
<script src="script.js"></script>
</html>
``
```

Creating the Header Section

Our header will contain a navigation bar and a menu icon which will trigger the sliding menu.

```
``html
<header>
  <div class="container">
    <h1>Menu Slider & Modal</h1>
    <button id="menu-btn">☰</button>
  </div>
</header>
``
```

Here:

- `container` will act as a wrapper to center our content.
- The `h1` provides a title.
- The `menu-btn` button will be used to trigger the sliding menu.

Crafting the Sliding Menu

The sliding menu will be a sidebar that slides in and out from the left. We'll start with the basic structure, and the sliding functionality will be added with JavaScript later on.

```
``html
<aside id="menu-slider">
  <ul>
    <li><a href="home">Home</a></li>
    <li><a href="services">Services</a></li>
    <li><a href="about">About</a></li>
    <li><a href="contact">Contact</a></li>
  </ul>
</aside>
``
```

Building the Modal Section

Our modal will be a simple pop-up that can be triggered via a button click. For the sake of this example, we'll use it to display a subscription form.

```
``html
<div id="modal">
  <div class="modal-content">
    <span id="close-btn">x</span>
    <h2>Subscribe to Our Newsletter</h2>
    <form action="">
      <input type="email" placeholder="Enter your email">
      <button type="submit">Subscribe</button>
    </form>
  </div>
</div>
``
```

In this structure:

- The outer `div` with id `modal` will act as a backdrop, dimming the rest of the page when the modal is visible.
- `modal-content` contains the actual contents of the modal.
- `close-btn` will be used to close the modal.
- The form within the modal prompts the user to subscribe to a newsletter.

Wrapping Up

With the HTML structure set for the Menu Slider & Modal project, you're ready to move on to styling with CSS and adding functionality with JavaScript. Always ensure your HTML is clear and semantic, as this not only makes your

website more accessible but also eases the process of styling and scripting.

In the next chapter, we will dive into styling our Navbar and setting up the overall look of our project with CSS.

Navbar Styling

Welcome to Chapter 35, where we'll be focusing on one of the most fundamental components of a web application's UI: the Navbar. The navigation bar, or navbar for short, typically contains links that help users navigate through different sections or pages of a website. Given its importance, styling the navbar is crucial for both aesthetics and usability. Let's dive into how we can style our navbar for the "Menu Slider & Modal" project.

1. The Basic Structure of a Navbar

Before we style, let's remind ourselves of the HTML structure we're working with. A typical navbar might look something like this:

```
``html
<nav class="navbar">
  <div class="logo">
    <h1>MyWebsite</h1>
  </div>
  <ul class="nav-menu">
    <li><a href="home">Home</a></li>
    <li><a href="about">About</a></li>
    <li><a href="services">Services</a></li>
    <li><a href="contact">Contact</a></li>
  </ul>
```

```
<div class="menu-toggle"><i class="fas fa-bars"></i>
</div>
</nav>
"
```

2. Styling the Navbar Container

To start, let's give our navbar a distinct look that separates it from the rest of the page.

```
"`css
```

```
.navbar {
    display: flex;
    justify-content: space-between;
    align-items: center;
    background-color: #333;
    padding: 10px 50px;
    box-shadow: 0px 3px 10px rgba(0, 0, 0, 0.2);
}
```

```
"`
```

- `display: flex;` ensures that the navbar's child elements are aligned in a row.

- `justify-content: space-between;` evenly distributes the child elements across the navbar.

- `box-shadow:` adds a subtle shadow, making the navbar appear slightly elevated from the page.

3. Styling the Logo

Our logo is the brand's representation; it should stand out but not overpower other elements.

```
"`css
```

```
.navbar .logo h1 {
```

```
        color: fff;  
        margin: 0;  
        font-size: 1.5rem;  
        text-transform: uppercase;  
        letter-spacing: 2px;  
    }  
``
```

4. Styling the Menu

Next, let's style the menu links. A modern practice is to use a horizontal list with spacing between items.

```
``css
```

```
.nav-menu {  
    list-style-type: none;  
    padding: 0;  
    display: flex;  
    gap: 20px;  
}  
.nav-menu li a {  
    color: fff;  
    text-decoration: none;  
    transition: color 0.3s ease;  
}  
.nav-menu li a:hover {  
    color: 007BFF; /* A shade of blue for hover effect */  
}  
``
```

5. Styling the Menu Toggle

This is the button users will click to see the menu on smaller screens:

```
``css
.menu-toggle {
    display: none; /* Initially hidden on desktop */
    font-size: 1.5rem;
    color: fff;
    cursor: pointer;
}

/* Media query to show the menu toggle on smaller screens */

@media (max-width: 768px) {
    .menu-toggle {
        display: block;
    }

    .nav-menu {
        display: none; /* Hide menu on mobile by default */
        flex-direction: column;
        gap: 10px;
        position: absolute;
        top: 60px; /* height of the navbar */
        left: 0;
        background-color: 333;
        width: 100%;
    }
}

.nav-menu.active { /* This class will be added via JS to show the menu */
    display: flex;
}
```

```
}
```

```
"
```

6. Additional Touches

To further refine our navbar, consider adding transitions or animations for a smoother user experience. For instance:

```
"`css
```

```
.nav-menu li a {  
    transition: all 0.3s ease;  
}
```

```
"
```

This will ensure that changes to properties like color, when hovering over the menu items, are gradual and smooth.

Conclusion

With these styles, our navbar is not only functional but also visually appealing. A well-styled navbar enhances user experience and guides them seamlessly through your application. As always, feel free to adjust colors, sizes, and other properties to better fit the overall design and theme of your website.

In the next chapter, we'll explore styling the header and modal components, diving deeper into the world of CSS and improving our project's look and feel. Stay tuned!

Header & Modal Styling

Welcome back! Having set up the basic structure for our Menu Slider & Modal project, it's time to give our header and modal some appealing visuals. Styling is a crucial part of any web project. It enhances the user experience and ensures your application is pleasant to the eyes. In

this chapter, we'll focus on styling the header and the modal using pure CSS.

1. Styling the Header

Our header typically contains the website's title and perhaps a navigation menu button. Let's make it stand out but still maintain a level of simplicity.

HTML Structure Reminder:

```
``html
<header>
  <h1>Our Website</h1>
  <button id="menu-btn">Menu</button>
</header>
``
```

CSS:

```
``css
/* Base Styling for the Header */
header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 1rem 2rem;
  background-color: #333;
  color: white;
  box-shadow: 0px 3px 10px rgba(0, 0, 0, 0.2);
}
header h1 {
  font-size: 2rem;
}
```

```
menu-btn {  
    background-color: 555;  
    border: none;  
    color: white;  
    padding: 0.5rem 1rem;  
    cursor: pointer;  
    border-radius: 5px;  
    transition: background-color 0.3s;  
}  
  
menu-btn:hover {  
    background-color: 777;  
}  
``
```

Here, we've used a flexbox layout for the header to space out the title and the button. The `box-shadow` gives a subtle elevation effect to the header.

2. Styling the Modal

The modal is a crucial part of this project. It should grab the user's attention, yet not be too distracting.

HTML Structure Reminder:

```
``html  
<div id="modal">  
    <div class="modal-content">  
        <span class="close-btn">&times;</span>  
        <h2>Modal Title</h2>  
        <p>Some content here...</p>  
    </div>  
</div>
```

```
```
CSS:
```
css
/* Base Styling for the Modal */
modal {
    display: none;
    position: fixed;
    top: 0;
    left: 0;
    width: 100vw;
    height: 100vh;
    background-color: rgba(0, 0, 0, 0.7);
    z-index: 1000;
    align-items: center;
    justify-content: center;
}
.modal-content {
    background-color: fff;
    padding: 2rem;
    width: 70%;
    max-width: 500px;
    border-radius: 10px;
    box-shadow: 0px 5px 20px rgba(0, 0, 0, 0.3);
}
.close-btn {
    position: absolute;
    top: 10px;
    right: 15px;
```

```
    font-size: 1.5rem;  
    cursor: pointer;  
    transition: color 0.3s;  
}  
.close-btn:hover {  
    color: red;  
}  
“
```

The outer `modal` div stretches across the whole viewport and has a semi-transparent background to give focus to the modal content. The ` `.modal-content` ` div holds our actual modal content and stands out against the dark background, thanks to its white color.

The ` `close-btn` ` is positioned in the top-right corner of the modal. We've also added a hover effect to provide feedback to the user.

Conclusion

With the header and modal now styled, our project is starting to take shape. Styling plays a vital role in dictating the feel and atmosphere of an application, and by now, you should have a better understanding of how to achieve this using pure CSS.

In the next chapter, we'll delve into the JavaScript behind the Menu and Modal Toggle functionalities, bringing our project to life!

Menu & Modal Toggle

In this chapter, we'll learn how to implement a toggle mechanism for our menu and modal using the Document Object Model (DOM) and Cascading Style Sheets (CSS). Toggles are frequently used in web design to show or hide content upon user interaction. They

enhance user experience by providing a dynamic interface.

1. The HTML Structure:

Before diving into the JavaScript and CSS for our toggle, let's set up the basic HTML structure:

```
``html
<div id="menu-container">
    <button id="menu-button">Menu</button>
    <div id="menu-content" class="hidden">
        <ul>
            <li><a href="">Home</a></li>
            <li><a href="">About</a></li>
            <li><a href="">Services</a></li>
        </ul>
    </div>
</div>
<button id="open-modal-btn">Open Modal</button>
<div id="myModal" class="modal hidden">
    <div class="modal-content">
        <span id="close-modal-btn" class="close">&times;</span>
        <p>Welcome to our website!</p>
    </div>
</div>
``
```

Note: We've added the class `hidden` to both the `menu-content` and `myModal` to hide them initially.

2. Styling the Toggle:

The `hidden` class is crucial as it will help in toggling the display of our menu and modal:

```
``css
.hidden {
    display: none;
}

.menu {
    /* Add general styling for the menu here */
}

.modal {
    position: fixed;
    z-index: 1;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    background-color: rgba(0,0,0,0.4);
}

.modal-content {
    background-color: fefefe;
    margin: 15% auto;
    padding: 20px;
    border: 1px solid 888;
    width: 80%;
}

``
```

3. The JavaScript Toggle Mechanism:

Using JavaScript, we can easily toggle the visibility of our menu and modal:

```
``javascript
document.addEventListener('DOMContentLoaded',
function() {

    const menuBtn = document.getElementById("menu-
button");

    const modalBtn = document.getElementById("open-
modal-btn");

    const closeModalBtn =
document.getElementById("close-modal-btn");

    const menuContent =
document.getElementById("menu-content");

    const myModal =
document.getElementById("myModal");

    menuBtn.addEventListener("click", function() {
        toggleVisibility(menuContent);
    });
    modalBtn.addEventListener("click", function() {
        toggleVisibility(myModal);
    });
    closeModalBtn.addEventListener("click", function() {
        toggleVisibility(myModal);
    });
    function toggleVisibility(element) {
        if (element.classList.contains('hidden')) {
            element.classList.remove('hidden');
        } else {
            element.classList.add('hidden');
        }
    }
})
```

```
    }  
});  
“
```

The above script listens for the `DOMContentLoaded` event, ensuring our JavaScript runs only after the entire HTML document has been completely loaded. We've defined a `toggleVisibility` function that toggles the visibility of an element by adding or removing the `hidden` class.

Conclusion:

By the end of this chapter, you've learned how to use JavaScript and CSS in tandem to create a toggle mechanism for a menu and a modal. By integrating this knowledge into your web projects, you can create a more interactive and dynamic user interface.

Section 8: Project 7 - Hangman Game | DOM, SVG, Events

Project Intro

Welcome to our seventh project - The Hangman Game! This classic word game not only tests your vocabulary but also provides a delightful user experience with the help of SVG for graphics, DOM for dynamic content manipulation, and JavaScript events to capture user inputs. By the end of this project, you'll have a fully functional Hangman game that you can show off to your friends or even include in your web portfolio.

Objective of the Game:

The aim of the Hangman game is straightforward. A word or phrase is chosen at random, and the player must guess the word letter by letter. For every wrong guess, a part of a “hangman” figure is drawn. The game ends when the figure is complete (indicating all guesses were wrong) or the word/phrase is entirely guessed correctly.

What You'll Learn:

- **SVG (Scalable Vector Graphics):** You'll understand how to use SVG to draw and animate the hangman figure. SVGs are a powerful way to add vector-based graphics to your webpage, and they can be manipulated using CSS and JavaScript.
- **DOM Manipulation:** As with our previous projects, the DOM plays a crucial role. You'll be updating the display based on user guesses, revealing letters, showing notifications, and more.
- **JavaScript Events:** User input is vital in the Hangman game. You'll be harnessing keyboard events to capture user guesses, button click events for game controls, and more.
- **Game Logic:** Behind every game lies a set of rules and logic. You'll be setting up conditions to check player guesses, determine when the game is won or lost, and more.

Why The Hangman Game?

Games have always been a fantastic way to learn coding. They introduce challenges that are slightly different from standard web development tasks and encourage thinking from both a developer and a player's perspective. Plus, games are engaging! They provide immediate feedback, making the coding and testing process fun.

Project Structure:

- We'll start by designing the game visually, setting up our main layout, and drawing our hangman using SVG.
- The primary styling will be set up, ensuring a responsive and engaging design.
- We'll introduce game mechanics, setting up the array of words/phrases and the functionality to select one at random.
- User inputs will be captured using JavaScript events.
- Based on user inputs, we'll provide feedback by revealing correct letters or drawing parts of the hangman for incorrect guesses.
- Additional features, such as popups and notifications, will further enhance our game, providing the player with instructions, feedback, or options to play again.

Prerequisites:

While this is a standalone project, it's beneficial if you have a basic understanding of HTML, CSS, and JavaScript, as we'll dive deep into some advanced topics. If you're entirely new, I recommend going through the previous sections of this book to build a solid foundation.

In conclusion, the Hangman game is an exciting project that blends fun with learning. As we progress through this section, remember to test your game frequently and enjoy the process. By the end of it, not only will you have a deeper understanding of web development techniques, but you'll also have a game to play and share. Let's get started!

Draw Hangman With SVG

SVG, or Scalable Vector Graphics, is an XML-based format for two-dimensional graphics. The beauty of SVG is that it can scale indefinitely without losing any quality, making it a popular choice for web graphics. In this chapter, we'll utilize SVG to draw the iconic hangman figure step by step.

Getting Started with SVG

SVG graphics are defined in XML, which means every element and attribute in the SVG is accessible via your JavaScript, allowing for dynamic creation and manipulation.

Here's a basic structure of an SVG:

```
``xml
<svg width="300" height="300"
xmlns="http://www.w3.org/2000/svg">
    <!-- SVG content goes here -->
</svg>
``
```

This defines an SVG canvas of 300x300 units. Anything drawn outside of this area will be clipped out.

Setting Up the Hangman Stand

Before drawing the hangman figure itself, let's set up the hangman stand.

```
``xml
<svg width="200" height="250"
xmlns="http://www.w3.org/2000/svg">
    <!-- Vertical line -->
    <line x1="60" y1="20" x2="60" y2="200"
style="stroke:black;stroke-width:5"/>
    <!-- Horizontal line -->
```

```
<line x1="10" y1="20" x2="150" y2="20"
style="stroke:black;stroke-width:5"/>
<!-- Small vertical line -->
<line x1="140" y1="20" x2="140" y2="40"
style="stroke:black;stroke-width:5"/>
</svg>
``
```

Drawing the Hangman with SVG

To break the drawing process into manageable steps, we'll draw the hangman piece by piece. For each incorrect guess, a new part of the hangman will appear.

1. Head

Using the `<circle>` SVG element:

```
``xml
```

```
<circle cx="140" cy="60" r="20" stroke="black" stroke-
width="3" fill="white" />
```

```
``
```

Here, `cx` and `cy` determine the circle's center, while `r` defines the radius.

2. Body

Using the `<line>` SVG element:

```
``xml
```

```
<line x1="140" y1="80" x2="140" y2="120"
style="stroke:black;stroke-width:3"/>
```

```
``
```

3. Left Arm

```
``xml
```

```
<line x1="140" y1="90" x2="120" y2="110"
style="stroke:black;stroke-width:3"/>
```

```
``
```

4. Right Arm

```
``xml
<line x1="140" y1="90" x2="160" y2="110"
style="stroke:black;stroke-width:3"/>
``
```

5. Left Leg

```
``xml
<line x1="140" y1="120" x2="120" y2="150"
style="stroke:black;stroke-width:3"/>
``
```

6. Right Leg

```
``xml
<line x1="140" y1="120" x2="160" y2="150"
style="stroke:black;stroke-width:3"/>
``
```

Dynamic Drawing with JavaScript

Instead of showing the entire figure right away, we want to draw each part in response to incorrect guesses. This means dynamically adding SVG elements using JavaScript.

Here's a sample code snippet to show the head after one incorrect guess:

```
``javascript
const svgNamespace = "http://www.w3.org/2000/svg";
function drawHead() {
    let circle =
        document.createElementNS(svgNamespace, "circle");
    circle.setAttribute("cx", "140");
    circle.setAttribute("cy", "60");
    circle.setAttribute("r", "20");
```

```
        circle.setAttribute("stroke", "black");
        circle.setAttribute("stroke-width", "3");
        circle.setAttribute("fill", "white");
        document.querySelector("svg").appendChild(circle);
    }
``
```

Repeat similar functions for the body, arms, and legs. Trigger the drawing functions based on the game's state and the number of incorrect guesses.

Conclusion

Drawing with SVG provides immense flexibility when creating graphics for web applications. In this chapter, we explored how to use SVG to dynamically draw the hangman figure for our game. By integrating SVG with JavaScript, we can make our game more interactive and responsive to player actions. As you progress with the Hangman game project, you'll appreciate the power and flexibility SVG brings to web development.

Main Styling

Welcome to Chapter 40 of our journey through creating a Hangman game. Now that we have a sketch of our hangman using SVG, it's time to give our game a visual appeal that is inviting and intuitive. We'll cover the main styling for our Hangman game, focusing on creating an engaging user experience.

1. Setting up the Basic Layout

Before diving into the specific elements, let's define some base styles to create a consistent look.

```
``css
* {
```

```
margin: 0;  
padding: 0;  
box-sizing: border-box;  
font-family: 'Arial', sans-serif;  
}  
  
body {  
background-color: f4f4f4;  
color: 333;  
font-size: 16px;  
line-height: 1.5;  
}  
  
.container {  
max-width: 800px;  
margin: 2em auto;  
background-color: fff;  
padding: 20px;  
box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);  
}  
``
```

This sets a neutral background color, a default font, and a centered container for our game.

2. Hangman Board Styling

Our hangman board will be the main attraction, so let's ensure it looks appealing.

```
``css  
.hangman-board {  
display: flex;  
justify-content: space-between;
```

```
margin-bottom: 20px;  
}  
.hangman-figure {  
    flex: 1;  
    display: inline-block;  
    position: relative;  
    width: 250px;  
    height: 250px;  
}  
“
```

3. Word Placeholder

The word that players are guessing will be displayed with underscores for each unguessed letter. We'll style these so they're easy to read and spaced nicely.

```
“css  
.word {  
    display: flex;  
    justify-content: center;  
    align-items: center;  
    height: 50px;  
    margin-bottom: 20px;  
}  
.letter, .placeholder {  
    margin: 0 5px;  
    font-size: 2em;  
    font-weight: bold;  
    display: inline-block;  
    width: 30px;
```

```
border-bottom: 2px solid 333;  
text-align: center;  
}  
“
```

4. Keyboard

For our game, players will select letters from an on-screen keyboard. Let's style these buttons:

```
“css  
.keyboard {  
    display: flex;  
    flex-wrap: wrap;  
    justify-content: center;  
}  
.key {  
    margin: 5px;  
    padding: 10px 15px;  
    border: none;  
    background-color: f4f4f4;  
    cursor: pointer;  
    border-radius: 4px;  
    transition: background-color 0.2s ease;  
}  
.key:hover {  
    background-color: ddd;  
}  
“
```

5. Notification Pop-up

We'll use a pop-up to notify players when they select a repeated letter.

```
``css
.notification {
    position: absolute;
    top: 10px;
    right: 10px;
    background-color: e74c3c;
    color: fff;
    padding: 10px 20px;
    border-radius: 4px;
    visibility: hidden;
    opacity: 0;
    transition: opacity 0.3s ease, visibility 0.3s ease;
}
.show-notification {
    visibility: visible;
    opacity: 1;
}```
```

6. Responsiveness

Finally, let's ensure our game looks good on smaller screens.

```
``css
@media screen and (max-width: 768px) {
    .hangman-board {
        flex-direction: column;
    }
}```
```

```
.hangman-figure, .word, .keyboard {  
    width: 100%;  
    text-align: center;  
}  
}  
“
```

With these styles in place, our Hangman game should now have a polished and user-friendly interface. In the next chapter, we will dive into creating pop-ups and notifications to guide our player through the game. Stay tuned!

Popup & Notification Styling

Welcome back to our exciting journey through the Hangman Game project! In this chapter, we're going to focus on the design and styling aspects of popups and notifications within our game. These elements are essential in creating an interactive user experience, guiding the player through their gameplay journey, and ensuring that they are informed about their progress or mistakes.

What We'll Cover

1. Understanding the importance of popups and notifications in a game.
2. Structuring the HTML for our popups and notifications.
3. Styling these elements using CSS for a smooth and interactive design.

1. Understanding the Importance

In any game, feedback is crucial. It allows players to understand what's going on, whether they're progressing, failing, or achieving something significant. Popups and notifications are classic ways of providing this feedback.

For our Hangman Game:

- Popups might be used to indicate game over scenarios, such as when the player wins or loses.
- Notifications will be handy for letting the player know if they've chosen a letter that's already been selected or if their selected letter isn't in the word.

2. HTML Structure

Before styling, we need a proper HTML structure in place. For simplicity, we'll create a basic modal for both our popup and notification.

```
``html
<!-- Popup Modal -->
<div class="popup-modal">
  <div class="popup-content">
    <span class="close-btn">&times;</span>
    <h2>Game Over!</h2>
    <p>You guessed the word correctly!
    Congratulations!</p>
  </div>
</div>
<!-- Notification Modal -->
<div class="notification-modal">
  <p>You've already chosen this letter. Try another
  one!</p>
</div>
``
```

3. Styling with CSS

Now, let's make these popups and notifications visually appealing and in line with the theme of our game.

Popup Modal Styling:

```
``css
```

```
.popup-modal {  
    display: none; /* Hidden by default */  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
    height: 100%;  
    background-color: rgba(0, 0, 0, 0.7); /* Semi-black  
background */  
    z-index: 1; /* Ensure it appears on top */  
}  
.popup-content {  
    background-color: fefefe;  
    margin: 15% auto; /* Centered vertically */  
    padding: 20px;  
    border-radius: 10px;  
    width: 70%;  
    box-shadow: 0 5px 15px rgba(0, 0, 0, 0.3);  
}  
.close-btn {  
    color: aaa;  
    float: right;  
    font-size: 28px;
```

```
    font-weight: bold;  
}  
.close-btn:hover,  
.close-btn:focus {  
    color: 000;  
    text-decoration: none;  
    cursor: pointer;  
}  
``
```

Notification Modal Styling:

```
``css  
.notification-modal {  
    background-color: ff6666; /* A reddish color indicating  
    an error */  
    color: fff;  
    position: fixed;  
    bottom: 20px;  
    left: 50%;  
    transform: translateX(-50%); /* Centering horizontally  
    */  
    padding: 10px 20px;  
    border-radius: 5px;  
    box-shadow: 0 2px 10px rgba(0, 0, 0, 0.2);  
    display: none; /* Hidden by default */  
}  
``
```

Bringing Them to Life with JavaScript:

To display the popups and notifications, we'll need to make use of the DOM.

For example, if a player chooses a repeated letter:

```
``javascript
const notificationModal =
document.querySelector('.notification-modal');

// Somewhere in your letter-checking logic
if (letterIsRepeated) {
    notificationModal.style.display = 'block';
    setTimeout(() => {
        notificationModal.style.display = 'none'; // Hide
        after 2 seconds
    }, 2000);
}
``
```

Similarly, for the popup modal, you'll toggle its visibility based on game events, such as game over scenarios.

Conclusion

And that wraps up our chapter on styling popups and notifications! With these elements in place, our Hangman Game will now be able to provide timely and visually appealing feedback to the player. Remember, the user experience is just as important as the core game logic. In our next chapter, we'll delve deeper into the game mechanics with "Display Words Function". Stay tuned!

Display Words Function

In the Hangman game, the core is to guess a word. The word can be random or selected from a list of predefined words. To make our game engaging, we will maintain a list of words and select a random word for the player to

guess. The `Display Words Function` will be responsible for presenting this word on the screen, masking the letters, and revealing them as the player makes correct guesses.

1. Setting up the Word Array

Before diving into the function, let's first create an array of possible words for our game:

```
``javascript
const words = ["javascript", "python", "typescript", "ruby",
"java"];
``
```

We've chosen a few programming languages for our Hangman game. Feel free to add as many words as you like.

2. Selecting a Random Word

We need to pick a random word from our `words` array. Let's create a function to achieve this:

```
``javascript
function getRandomWord() {
    return words[Math.floor(Math.random() *
words.length)];
}
``
```

Here, `Math.random()` gives a random number between 0 and 1, and multiplying it with the length of the `words` array will give a number between 0 and the array length. `Math.floor()` ensures we get an integer value which is used as an index to fetch a word.

3. Display Words Function

The primary goal of this function is to display the word on the game board with the letters masked (typically using underscores “_”).

Let's define our function:

```
``javascript
let selectedWord = getRandomWord();
let displayWord = [];
function displayWordsFunction() {
    displayWord = selectedWord.split("").map(letter =>
    '_');
    updateDisplay();
}
``
```

Here, we're breaking the `selectedWord` into individual letters using the `split()` method and then using the `map()` function to replace each letter with an underscore.

4. Update Display Function

This function will be responsible for updating the word display on our HTML:

```
``javascript
function updateDisplay() {
    const displayElement =
document.getElementById('wordDisplay');
    displayElement.innerHTML = displayWord.join(' ');
}
``
```

Make sure you have an element with the id `wordDisplay` in your HTML to display the word.

5. Revealing the Guessed Letters

As the player makes correct guesses, we need to reveal the corresponding letters in the word. Let's add this functionality:

```
``javascript
function revealLetter(letter) {
    selectedWord.split("").forEach((char, index) => {
        if (char === letter) {
            displayWord[index] = char;
        }
    });
    updateDisplay();
}
``
```

The `revealLetter` function checks each character in the `selectedWord`. If it matches the guessed letter, it updates the `displayWord` array at the respective index.

6. Final Touches

To initialize our game, let's call the `displayWordsFunction()` when the game starts or resets:

```
``javascript
function startGame() {
    selectedWord = getRandomWord();
    displayWordsFunction();
    // any other game initialization code...
}
``
```

Conclusion

Our `Display Words Function` is the cornerstone of our Hangman game. It selects a random word, masks it for the player, and reveals the letters as they are guessed correctly. Coupled with the game's logic and user interface, it forms an engaging experience for players. In the subsequent chapters, we will delve into handling user input and adding more features to our game.

Letter Press Event Handler

Handling events is a fundamental aspect of interactive web applications. In our Hangman game, detecting a user's letter input is pivotal to the game's functionality. This chapter focuses on the Letter Press Event Handler that facilitates this interaction. Let's dive into how to set it up and use it effectively within our game.

Event Handlers in JavaScript

Before delving into the specifics of the Letter Press Event Handler, it's important to understand event handlers in JavaScript. An event handler is a function that runs when a specific event occurs. For instance, when a button is clicked, a certain function can be set to run. The process of setting a function to run in response to an event is called **binding** an event to an element.

Setting Up the Letter Press Event Handler

1. HTML Structure: Ensure that your Hangman game has an interactive section, likely an input box or a series of buttons, where players can enter or select letters.

```
``html
<div id="letter-box">
    <button class="letter">A</button>
    <button class="letter">B</button>
```

```
<!—... repeat for all letters —>
</div>
“
2. JavaScript Event Binding: To bind the press event to each letter, we'll utilize the `addEventListener` method.

``javascript
document.querySelectorAll('.letter').forEach(letterButton => {
    letterButton.addEventListener('click', handleLetterPress);
});
``
```

Implementing the `handleLetterPress` Function

Here's where the magic happens. When a letter is clicked, the `handleLetterPress` function is triggered.

```
``javascript
function handleLetterPress(event) {
    // Extract the letter from the clicked button
    const chosenLetter = event.target.innerText;
    // Check if the chosen letter exists in the word to be guessed
    if (wordToGuess.includes(chosenLetter)) {
        // ... reveal the letter in the display
    } else {
        // ... update wrong guesses and potentially draw a part of the hangman
    }
    // Disable the button to prevent repeated guesses
    event.target.disabled = true;
```

```
}
```

```
"
```

Understanding Event Object

In the above code, we used the `event` parameter. This parameter gives us access to the event object, which contains details about the event, such as which element triggered it, the type of event, and more.

- `event.target`: Refers to the element that triggered the event. In our case, it's the button representing a letter.

Enhancing the Experience

- Key Presses: Instead of clicking on buttons, you can also let users type in their guesses. Use the `keydown` event for this:

```
"`javascript
```

```
document.addEventListener('keydown', function(event) {  
    // Ensure the key pressed is a valid letter and hasn't  
    // been guessed already
```

```
    if (isLetter(event.key) && !isGuessed(event.key)) {
```

```
        handleLetterPress(event.key.toUpperCase());
```

```
}
```

```
});
```

```
"`
```

- Feedback: Provide immediate feedback to the user. Maybe the background of the letter button changes color based on whether the guess was correct or incorrect.

Conclusion

The Letter Press Event Handler is crucial in allowing interaction in our Hangman game. It helps capture user input and makes the game responsive and dynamic. By

understanding how event handlers work in JavaScript and how to effectively bind and handle them, you've taken a significant step in enhancing user interactivity in web applications.

Remember, while our focus was on the Hangman game, the principles of event handling are universally applicable in JavaScript and are key to creating interactive web applications.

In the next chapter, we'll explore how to handle wrong guesses, update the display to draw parts of the hangman, and offer players a chance to play again once the game concludes. Stay tuned!

Wrong Letters & Play Again

Welcome back to our Hangman Game project! In this chapter, we'll focus on two crucial parts of the game's functionality: handling wrong letters and providing a way for players to play again once the game concludes.

1. Setting the Foundation

Before diving into the code, let's understand the two functionalities:

- Wrong Letters: When a user selects a letter that isn't in the word, we need a way to indicate this to the user. This often comes in the form of updating the hangman drawing and notifying the user of their incorrect choice.
- Play Again: After the user either wins or loses the game, we should provide an option to play again without having to refresh the page.

2. Handling Wrong Letters

2.1 Updating the Hangman Drawing

For this project, we're using SVG to draw the hangman. Every incorrect guess will result in an additional part of

the hangman being drawn.

Let's set this up in our JavaScript:

```
``javascript
const figureParts = document.querySelectorAll('.figure-part');

let wrongLetters = [];

function updateHangmanDrawing() {
    figureParts.forEach((part, index) => {
        const errors = wrongLetters.length;
        if (index < errors) {
            part.style.display = 'block';
        } else {
            part.style.display = 'none';
        }
    });
}
```

“

This function will show parts of the hangman based on the number of wrong letters.

2.2 Displaying Wrong Letter Choices

We should also notify the user of the letters they've chosen that are incorrect. This provides feedback and avoids repetition of the same incorrect letter.

```
``javascript
const wrongLetterEl =
document.getElementById('wrong-letters');

function updateWrongLettersEl() {
    wrongLetterEl.innerHTML =
${wrongLetters.length > 0 ? '<p>Wrong</p>' : ""}
```

```
        ${wrongLetters.map(letter => `<span>${letter}</span>`).join(', ')}

    ;
    updateHangmanDrawing();
}

``
```

Every time a player inputs a wrong letter, this function will update a dedicated element on the page to show those letters.

3. Play Again Functionality

3.1 Play Again Button

First, let's add a simple button that will appear when the game concludes.

```
``html
<div id="play-again-btn" class="popup" style="display: none;">
    <h2>Play Again?</h2>
    <button id="play-button">Start Over</button>
</div>
``
```

3.2 Implementing the Play Again Logic

Now, in our JavaScript, let's reset the game when this button is clicked:

```
``javascript
const playAgainBtn = document.getElementById('play-button');

playAgainBtn.addEventListener('click', () => {
    // Empty the wrongLetters array
    wrongLetters.splice(0);
```

```
// Reset game status
finalMessage.innerText = "";
popup.style.display = 'none';

// Reset UI elements - wrong letters, hangman
drawing
updateWrongLettersEl();

// ... any other game reset functionality you have

// Start a new game instance
initGame();

});

``
```

Remember to also create and define the `initGame` function, which will be responsible for starting a fresh game instance.

4. Wrapping Up

With these functionalities in place, the Hangman game becomes interactive and user-friendly. Handling wrong letter choices appropriately and providing feedback improves the user experience, while the Play Again option keeps players engaged.

Section 9: Project 8 - Meal Finder | Fetch & MealDB API

Project Intro

Welcome to Project 8, the Meal Finder application! This particular project is an exciting journey into the world of API interaction and displaying dynamic content on the web. As the name suggests, our goal here is to build a web-based application that helps users find meals based

on a specific query and explore detailed information about them. Here's what you can expect from this project:

Overview

In this project, we'll be developing a Meal Finder application, where users can search for meals, get a list of matching results, view a single meal's details, and even receive a random meal suggestion. All of this will be powered by the MealDB API, a free-to-use database containing meal recipes and details.

Why This Project?

1. Understanding APIs: APIs (Application Programming Interfaces) allow different software to communicate and share data. In the world of web development, knowing how to work with APIs is a crucial skill, and this project offers a practical application of it.
2. Deep Dive into Fetch: The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. We will be using the Fetch API extensively to retrieve data from the MealDB.
3. Dynamic UI Updates: This project will give you hands-on experience in updating the User Interface dynamically based on the data fetched from an external source.

What Will We Build?

1. Search Functionality: Users can input meal names or ingredients to search for matching meals.
2. Display Meals: We'll showcase the meals in a grid format, providing a visual treat for the user.
3. Detailed View: When a user clicks on a meal, they can view detailed information about it, including ingredients, preparation steps, and even a video tutorial.

4. Random Meal Suggestion: Add a fun feature where users can get a random meal suggestion, helping them discover new recipes.

Technologies & Techniques

- HTML & CSS: Our trusted companions. We'll use them for creating and styling our web page.
 - JavaScript: This will be the backbone of our project. We'll use it for fetching data, manipulating the DOM, and adding interactivity.
 - Fetch API: For making requests to the MealDB API and fetching data.
 - Async/Await: A modern way to handle asynchronous operations in JavaScript. We'll utilize this for a more readable and clean code structure when working with the Fetch API.
 - JSON: Data fetched from the API will be in JSON format. We'll parse this data and utilize it to update our webpage dynamically.
-

Challenges Ahead

1. Handling API Responses: Not every search query will return results. Learning how to handle such scenarios gracefully enhances the user experience.
2. Dynamic DOM Manipulation: Based on the data fetched, our application's appearance will change dynamically.
3. Error Handling: APIs can sometimes fail, or there might be issues with connectivity. Proper error handling ensures that our application is robust and user-friendly.

In the following chapters, we'll delve into the nitty-gritty details of building the Meal Finder application. From setting up the basic structure in HTML to styling our components with CSS and finally bringing everything to

life with JavaScript. So, roll up your sleeves and let's get cooking!

Note: Always refer to the MealDB API documentation while working on this project. It provides valuable insights into request endpoints, response formats, and more. Always respect the API usage guidelines and terms of service.

Project HTML & Base CSS

In this chapter, we'll lay the foundation for our 'Meal Finder' project. By the end, you'll have a visually appealing HTML structure complemented by base CSS styling. This will serve as the foundation upon which we will interact with the MealDB API in the subsequent chapters.

Step 1: Setting up the HTML structure

For our Meal Finder app, we'll need an input field for searching, a button to trigger the search, a container to display our results, and an area to show detailed information about a selected meal.

```
``html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Meal Finder</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
```

```
<div class="container">
  <h1>Meal Finder</h1>
  <div class="search-box">
    <input type="text" id="search-input"
placeholder="Search for a meal...">
    <button id="search-btn">Search</button>
  </div>
  <div class="meal-list"></div>
  <div class="meal-details"></div>
</div>
</body>
</html>
```

Step 2: Adding the Base CSS

For our base styling, we'll make use of modern CSS techniques, ensuring our app looks sleek and is also responsive.

```
*styles.css*:
``css
body {
  font-family: 'Arial', sans-serif;
  background-color: f4f4f4;
  margin: 0;
  padding: 0;
}
.container {
  width: 80%;
  margin: 2rem auto;
```

```
background-color: fff;
padding: 1rem;
box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h1 {
    text-align: center;
    color: 333;
    margin-bottom: 1rem;
}

.search-box {
    display: flex;
    justify-content: center;
    gap: 1rem;
}

search-input {
    padding: 0.5rem 1rem;
    font-size: 1rem;
    border: 1px solid ccc;
    border-radius: 5px;
    flex: 2;
}

search-btn {
    padding: 0.5rem 1rem;
    background-color: 007BFF;
    color: fff;
    border: none;
    border-radius: 5px;
    cursor: pointer;
```

```
        transition: background-color 0.3s;  
    }  
    search-btn:hover {  
        background-color: #0056b3;  
    }  
.meal-list, .meal-details {  
    margin-top: 2rem;  
}  
``
```

Summary

At this stage, our ‘Meal Finder’ project now has a basic layout. With our structured HTML and base CSS styling, we have a clean and intuitive interface ready. In the next chapters, we will integrate the functionality to search for meals using the MealDB API, display the results in our app, and allow users to view detailed information about each meal.

Search & Display Meals From API

In this chapter, we will delve deep into the world of APIs, specifically focusing on the MealDB API. We’ll take a look at how to integrate it into our Meal Finder project, allowing users to search for meals and view the details. By the end of this chapter, you’ll have hands-on experience with fetching data from an API and rendering it dynamically to the DOM using vanilla JavaScript.

Understanding the MealDB API

Before diving into the code, let’s familiarize ourselves with the MealDB API. This API provides recipe meal

information and is free to use. You can search for meals, look up specific meals by their ID, and even fetch random meal suggestions.

Endpoint for searching meals by name:

``

<https://www.themealdb.com/api/json/v1/1/search.php?s={meal-name}>

``

Replace `'{meal-name}'` with the meal you want to search for. The returned JSON will contain an array of meals that match the search.

Setting up the Search Functionality

1. HTML Structure

We'll need an input field for the user to type in their meal name and a button to trigger the search.

``html

```
<input type="text" id="meal-search" placeholder="Search for meals...">
<button id="search-btn">Search</button>
<div id="meal-results"></div>
```

``

2. JavaScript - Fetching the Data

To get the data from the API, we'll use the Fetch API in JavaScript. We'll attach an event listener to our search button to trigger the function.

``javascript

```
const searchBtn = document.getElementById('search-btn');
const mealSearch = document.getElementById('meal-search');
```

```
const mealResults = document.getElementById('meal-results');

searchBtn.addEventListener('click', fetchMeals);

function fetchMeals() {
    const searchTerm = mealSearch.value.trim();
    fetch(`https://www.themealdb.com/api/json/v1/1/search.php?s=${searchTerm}`)
        .then(response => response.json())
        .then(data => displayMeals(data.meals));
}

``
```

Displaying the Results

Once we get the meal data, we need to display it. We'll do this by iterating over the array of meals and creating a card for each one.

```
``javascript
function displayMeals(meals) {
    if(!meals) {
        mealResults.innerHTML = '<p>No meals found.  
Try a different search!</p>';
        return;
    }
    const mealsHTML = meals.map(meal => `
        <div class="meal-card">
            
            <h3>${meal.strMeal}</h3>
            <button
            onclick="fetchMealDetails(${meal.idMeal})">View
            Recipe</button>
    `);
    mealResults.innerHTML = mealsHTML;
}
```

```
        </div>
      `).join(""));
    mealResults.innerHTML = mealsHTML;
  }
``
```

This function checks if there are any meals in the returned array. If not, it shows a “No meals found” message. Otherwise, it maps over the meals array, generating HTML for each meal, and inserts it into the DOM.

Fetching Detailed Meal Information

Our cards have a “View Recipe” button. When clicked, we want to fetch more detailed information about the meal. This requires another endpoint from the MealDB API, which retrieves details based on meal ID.

Endpoint for meal details by ID:

“

<https://www.themealdb.com/api/json/v1/1/lookup.php?i={meal-id}>

“

The function to fetch and display the details could look like:

```
“javascript
function fetchMealDetails(id) {
  fetch(`https://www.themealdb.com/api/json/v1/1/lookup.php?i=${id}`)
    .then(response => response.json())
    .then(data => {
      const meal = data.meals[0];
      const detailsHTML = `
```

```
<h2>${meal.strMeal}</h2>

<p>${meal.strInstructions}</p>
`;
mealResults.innerHTML = detailsHTML;
});
}
``
```

This function fetches the detailed meal data using the provided ID, and then renders it to the DOM.

Conclusion

By now, you should have a functional Meal Finder that allows users to search for meals using the MealDB API. The results are displayed dynamically, and users can view detailed recipes with just a click. Remember, APIs offer a plethora of data that can be used to enhance your web applications. The key lies in understanding the API documentation, structuring your fetch requests appropriately, and presenting the data to users in a user-friendly manner.

Show Single Meal Page

Welcome back! In the last chapter, we saw how to fetch meals and display them based on user queries. Now, we are going to focus on how to show the details of a selected meal on a dedicated page. By the end of this chapter, you will have an interactive page where users can click on a meal and view its details including ingredients and preparation steps.

Objective:

- Fetch detailed data for a single meal using the MealDB API.
 - Display the meal details including image, name, category, origin, ingredients, and preparation steps.
-

Getting Started:

To show the single meal page, we will be utilizing the `idMeal` property which is unique to each meal and can be used to fetch the detailed information of that meal from the MealDB API.

1. HTML Structure:

Let's first design the layout for our single meal page.

```
``html
```

```
<div class="single-meal">
  <div class="single-meal-header">
    <img src="" alt="Meal Image" id="meal-img">
    <h2 id="meal-name"></h2>
  </div>
  <div class="single-meal-info">
    <p><strong>Category:</strong> <span id="meal-category"></span></p>
    <p><strong>Origin:</strong> <span id="meal-origin"></span></p>
  </div>
  <div class="single-meal-ingredients">
    <h3>Ingredients:</h3>
    <ul id="ingredients-list"></ul>
  </div>
  <div class="single-meal-instructions">
    <h3>Instructions:</h3>
```

```
<p id="meal-instructions"></p>
</div>
</div>
"
```

2. CSS Styling:

For a better presentation, add some CSS for the single meal display.

```
"`css
```

```
.single-meal-header {
    display: flex;
    align-items: center;
}

meal-img {
    width: 150px;
    height: 150px;
    margin-right: 20px;
    object-fit: cover;
    border-radius: 50%;
}

.single-meal-info {
    margin-top: 20px;
}

.single-meal-ingredients, .single-meal-instructions {
    margin-top: 30px;
}
```

3. JavaScript Functionality:

We will fetch and display the single meal's details using JavaScript.

```
``javascript
function getSingleMealDetails(mealID) {
    fetch(`https://www.themealdb.com/api/json/v1/1/lookup.php?i=${mealID}`)
        .then(response => response.json())
        .then(data => {
            const meal = data.meals[0];
            displayMealDetails(meal);
        })
        .catch(error => console.error('Error fetching single meal:', error));
}

function displayMealDetails(meal) {
    // Clear previous data
    document.getElementById('meal-img').src =
    meal.strMealThumb;
    document.getElementById('meal-name').textContent =
    meal.strMeal;
    document.getElementById('meal-
    category').textContent = meal.strCategory;
    document.getElementById('meal-origin').textContent =
    meal.strArea;
    // List ingredients
    const ingredientsList =
    document.getElementById('ingredients-list');
    ingredientsList.innerHTML = "";
    for(let i = 1; i <= 20; i++) {
        if(meal[`strIngredient${i}`]) {
```

```
const li = document.createElement('li');
    li.textContent = `${meal['strIngredient${i}`]} - 
${meal['strMeasure${i}`]}`;
    ingredientsList.appendChild(li);
}
}

// Display instructions
document.getElementById('meal-
instructions').textContent = meal.strInstructions;
}

``
```

4. Event Handling:

When a user clicks on a meal from the list, fetch its details:

```
``javascript
document.body.addEventListener('click', (e) => {
    if (e.target.classList.contains('meal-item')) {
        const mealID = e.target.getAttribute('data-mealid');
        getSingleMealDetails(mealID);
    }
});

``
```

Make sure that each meal displayed in the list has the class `meal-item` and a `data-mealid` attribute with the meal's unique ID.

Wrapping Up:

You've successfully created a single meal page that fetches detailed information about a meal and displays it in a user-friendly format. Now, when users are intrigued

by a meal's name or picture, they can dive deeper into its ingredients and instructions.

In the next chapter, we will delve into how to display random meals and style their presentation. Stay tuned!

Display Random Meal & Single Meal CSS

Welcome back to Project 8: Meal Finder. In our previous chapters, we've covered fetching meals from the MealDB API and presenting them in a list format. Now, we're going to dive deeper into our project by styling our Single Meal display and creating an attractive view for randomly presented meals.

Why Focus on Styling?

The UI (User Interface) is one of the most crucial parts of any web application. It determines how users interact with the application and influences their overall experience. A good UI should be intuitive and visually appealing. By providing a captivating design for our meals, we're aiming to create a memorable experience for our users, which will hopefully encourage them to use our application more frequently.

Display Random Meal CSS

When we display a random meal, our aim is to make it the centerpiece of the screen, giving it the emphasis it deserves.

HTML Structure:

```
“html
<div class=“random-meal”>
    <img src=“meal-image.jpg” alt=“meal-name”
        class=“random-meal-image”>
```

```
<div class="random-meal-details">
    <h2 class="random-meal-title">Meal Name</h2>
    <p class="random-meal-description">Short
description...</p>
</div>
</div>
``
```

CSS Styling:

```
``css
.random-meal {
    display: flex;
    align-items: center;
    margin: 20px 0;
    border-radius: 8px;
    overflow: hidden;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}

.random-meal-image {
    width: 150px;
    height: 150px;
    object-fit: cover;
}

.random-meal-details {
    flex: 1;
    padding: 10px;
}

.random-meal-title {
    font-size: 24px;
```

```
    font-weight: bold;  
    margin-bottom: 10px;  
}  
.random-meal-description {  
    font-size: 16px;  
    color: 555;  
}  
“
```

Single Meal CSS

The single meal display will provide more details about a particular meal, from its ingredients to its preparation steps.

HTML Structure:

```
“`html  
<div class=“single-meal”>  
    <img src=“meal-image.jpg” alt=“meal-name”  
        class=“single-meal-image”>  
    <div class=“single-meal-details”>  
        <h2 class=“single-meal-title”>Meal Name</h2>  
        <ul class=“ingredients-list”>  
            <li>Ingredient 1</li>  
            <!-- ... -->  
        </ul>  
        <p class=“single-meal-instructions”>Cooking  
            instructions...</p>  
    </div>  
</div>  
“`
```

CSS Styling:

```
``css
.single-meal {
    display: flex;
    margin: 30px 0;
    border-radius: 8px;
    overflow: hidden;
    box-shadow: 0 4px 15px rgba(0, 0, 0, 0.15);
}

.single-meal-image {
    width: 300px;
    height: 300px;
    object-fit: cover;
}

.single-meal-details {
    flex: 1;
    padding: 20px;
    display: flex;
    flex-direction: column;
    gap: 20px;
}

.single-meal-title {
    font-size: 32px;
    font-weight: bold;
    margin-bottom: 15px;
}

.ingredients-list {
    list-style-type: none;
```

```
padding: 0;  
}  
.ingredients-list li {  
    font-size: 18px;  
    color: 333;  
    margin: 5px 0;  
}  
.single-meal-instructions {  
    font-size: 18px;  
    color: 666;  
}  
``
```

Conclusion

With the above CSS, we've successfully enhanced the visual representation of our random and single meal displays. A well-structured and visually appealing design will not only make our application look professional but also provide a pleasant experience for our users.

Section 10:

Project 9 - Expense Tracker |
Array Methods & Local
Storage

Project Intro

As we journey through the vast world of web development, we've seen how the amalgamation of HTML, CSS, and JavaScript can give life to a variety of dynamic applications. From form validation to creating interactive movie seat bookings, every project has added a new feather to your web development cap. In this project, we'll delve into an essential tool that many individuals use daily - an Expense Tracker. By the end of this project, you'll have built a dynamic application that not only tracks expenses and income but also persists data using local storage.

Project Overview

In today's digital age, where the proliferation of mobile applications has taken center stage, expense tracking applications have seen a massive rise in popularity. They help users monitor their spending habits, keep track of savings, and provide insights into their financial health. With the Expense Tracker project, we aim to equip you with the skills to build your own version of this handy tool.

The primary objective of our Expense Tracker is:

- To record and display transactions (both expenses and income).
- To display the current balance, total income, and total expenditure.
- To provide functionalities to add and delete transactions.
- To store transaction data in the local storage, ensuring data persistence even after refreshing the page.

Why This Project?

You might be wondering, why an Expense Tracker?

1. Practical Utility: The Expense Tracker is a practical tool that many users would find beneficial in their daily

lives. Building applications that have real-world utility can be rewarding and motivating.

2. Complexity: This project strikes a balance between simplicity and complexity. It's comprehensive enough to challenge you but not overwhelmingly intricate that it becomes a hurdle.

3. Diverse Skills Application: This project provides an excellent platform to apply various skills like manipulating the DOM, using high order array methods like `forEach`, `map`, `filter`, and `reduce`, and leveraging the local storage.

Technologies & Concepts Used

- HTML & CSS: Foundation of our Expense Tracker's frontend. You'll be creating structured HTML templates and styling them to make the application visually appealing.

- JavaScript: The brain behind the operation. JS will give life to the application, allowing users to interact with the Expense Tracker.

- DOM Manipulation: You'll extensively manipulate the DOM to display transactions, update balance, income, and expenses.

- Array Methods: This project will make use of high order array methods, which are fundamental when working with lists of data in JS. Expect to use methods like `map` for processing and `filter` for displaying specific transactions.

- Local Storage: One of the most exciting aspects of this project. You'll ensure that data entered by the user persists even after the browser session ends.

Conclusion

Brace yourself for an exhilarating journey as we navigate through the intricacies of building an Expense Tracker.

By the end of this project, you'll have a functional tool at your disposal and a bolstered understanding of how to build practical applications from scratch.

Now, let's dive into the HTML structure in the next chapter, setting the foundation for our Expense Tracker!

Project HTML

Welcome to the first technical step in building our Expense Tracker! HTML is the backbone of any web application. This chapter will help you structure the Expense Tracker, ensuring that all the following steps in CSS and JavaScript can be implemented seamlessly.

1. Introduction

In this chapter, we'll lay out the basic HTML structure for our Expense Tracker application. This will include areas to display the balance, income, expenses, a list of transactions, and a form to add new transactions. The goal is to ensure the markup is semantic, accessible, and easy to style.

2. Setting up the Document

Start with the basic HTML boilerplate.

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Expense Tracker</title>
```

```
<!-- Link to your CSS file (you'll create this in the  
next chapter) -->  
<link rel="stylesheet" href="styles.css">  
</head>  
<body>  
<!-- Your content will go here -->  
</body>  
</html>  
“
```

3. Main Container

Let's start by creating the main container for our application.

```
“html  
<div class="container">  
    <!-- All other components will go here -->  
</div>  
“
```

4. Header and Balance

We'll display the name of the application and the user's balance at the top.

```
“html  
<h2>Expense Tracker</h2>  
<div class="balance">  
    <h3>Your Balance</h3>  
    <h4 id="balance-amount">$0.00</h4>  
</div>  
“
```

5. Income and Expense Summary

Below the balance, we'll have two containers displaying the total income and total expenses.

```
``html
<div class="inc-exp-container">
    <div>
        <h4>Income</h4>
        <p id="money-plus" class="money plus">+$0.00</p>
    </div>
    <div>
        <h4>Expense</h4>
        <p id="money-minus" class="money minus">-$0.00</p>
    </div>
</div>
``
```

6. Transaction History

This section will display the list of transactions added by the user.

```
``html
<h3>History</h3>
<ul id="transactions-list" class="transactions">
    <!-- Individual transactions will be added here
    dynamically -->
</ul>
``
```

7. Add New Transaction

Finally, we need a form for users to input new transactions.

```
``html
<h3>Add new transaction</h3>
<form id="form">
  <div class="form-control">
    <label for="text">Text</label>
    <input type="text" id="text" placeholder="Enter text...">
  </div>
  <div class="form-control">
    <label for="amount">Amount</label>
    <input type="number" id="amount" placeholder="Enter amount...">
    <small>(negative - expense, positive - income)</small>
  </div>
  <button class="btn">Add transaction</button>
</form>
``
```

8. Conclusion

With this structure in place, we've laid the groundwork for our Expense Tracker. The next steps will involve styling with CSS to make it visually appealing and then adding interactivity with JavaScript. Remember, a strong foundation with HTML is crucial, as it ensures our application is both functional and accessible.

Project CSS

In this chapter, we'll be diving deep into the styling of our "Expense Tracker" project. CSS, short for Cascading Style Sheets, is the language we use to design our web applications. It controls everything from the layout to the fonts, colors, and animations, creating a compelling user interface that's both functional and aesthetically pleasing.

1. Why CSS is Important

Before we begin coding, it's crucial to understand the significance of CSS. For our Expense Tracker, a clean and organized UI ensures users can easily add and monitor their expenses without any distractions. Styling also helps to reinforce functionality – by making important buttons or warnings prominent, we guide the user's experience in a positive direction.

2. Setting Up the Base Style

For consistency across browsers, we'll start by resetting some of the default browser styles:

```
``css
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
    font-family: 'Arial', sans-serif;
}
``
```

3. Styling the Main Container

Our Expense Tracker app will be contained within a main div. We want this container to be centered and have some space around it for a pleasant look:

```
``css
.container {
    width: 80%;
    max-width: 600px;
    margin: 40px auto;
    padding: 20px;
    background-color: f4f4f4;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}
``
```

4. Header Styles

A bold header will help users quickly identify the app:

```
``css
header {
    text-align: center;
    margin-bottom: 20px;
}
header h2 {
    font-size: 2rem;
    color: 333;
}
``
```

5. Input Fields and Buttons

Our app will have input fields for users to add expenses and buttons to submit them. These elements should be easy to spot and use:

```
``css
input, button {
    width: 100%;
    padding: 10px;
    margin-bottom: 10px;
    border: 1px solid ccc;
    border-radius: 4px;
    font-size: 1rem;
}

button {
    background-color: 007BFF;
    color: fff;
    cursor: pointer;
}

button:hover {
    background-color: 0056b3;
}

``
```

6. Displaying Transactions

Expenses will be listed in rows. Positive values (income) will be displayed in green, while negative values (expenses) will be in red for easy differentiation:

```
``css
.transaction {
    display: flex;
    justify-content: space-between;
    padding: 10px;
    border-bottom: 1px solid ddd;
```

```
}

.transaction.income {
    color: green;
}

.transaction.expense {
    color: red;
}

``
```

7. Totals, Income, and Expense Display

It's vital for users to get a quick overview of their financial status:

```
``css

.overview {
    display: flex;
    justify-content: space-between;
    margin-top: 20px;
}

.total, .income, .expense {
    width: 30%;
    padding: 10px;
    border: 1px solid ddd;
    border-radius: 4px;
    text-align: center;
}

.total {
    color: 333;
}

.income {
```

```
    color: green;  
}  
.expense {  
    color: red;  
}  
``
```

8. Responsive Design

Given the increasing number of users accessing apps from mobile devices, we must ensure our design looks great on smaller screens:

```
``css  
@media (max-width: 480px) {  
    .container {  
        width: 95%;  
    }  
    .overview {  
        flex-direction: column;  
    }  
    .total, .income, .expense {  
        width: 100%;  
        margin-bottom: 10px;  
    }  
}
```

Summary

The styling of our “Expense Tracker” app emphasizes clarity and user-friendliness. We’ve used a combination of layouts, color schemes, and responsive design to

ensure our application looks and functions well across various devices. In the next chapters, we'll dive into the functionality of our app, where we'll employ JavaScript to bring our Expense Tracker to life.

Show Transaction Items

In this chapter, we'll be focusing on displaying the transaction items in our Expense Tracker project. These transaction items represent individual financial activities, either income or expenditure, that the user has recorded in the app. We'll utilize a combination of HTML, CSS, and JavaScript to achieve this. Let's dive in!

1. The HTML Structure

To display transaction items, we need a container in our HTML where each transaction can be listed. The following markup will serve our purpose:

```
``html
<section class="transactions">
  <h2>Transaction History</h2>
  <ul id="transaction-list">
    <!-- Transaction items will be added here using
    JavaScript -->
  </ul>
</section>
``
```

Here, the `transaction-list` will be dynamically populated with the transaction items using JavaScript.

2. CSS Styling

To make our transaction list visually appealing, let's style it:

```
``css
.transactions {
    margin: 20px 0;
    border: 1px solid e0e0e0;
    padding: 10px;
}

.transactions h2 {
    text-align: center;
    margin-bottom: 15px;
}

transaction-list {
    list-style-type: none;
    padding: 0;
}

transaction-list li {
    display: flex;
    justify-content: space-between;
    padding: 8px 0;
    border-bottom: 1px solid e0e0e0;
}

``
```

3. JavaScript Implementation

First, select the transaction list in your JavaScript:

```
``javascript
const transactionList =
document.getElementById('transaction-list');

``
```

Assuming you have an array of transaction objects named `transactions` (which will be filled with user input or data from local storage), we will use the following function to display them:

```
``javascript
function displayTransactions() {
    // Clear out any current transaction items in the list
    transactionList.innerHTML = "";

    // Loop through the transactions array and display
    // each one
    transactions.forEach(transaction => {
        const item = document.createElement('li');
        item.innerHTML = `
            <span>${transaction.description}</span>
            <span>${transaction.amount > 0 ? '+' :
            '-'}${Math.abs(transaction.amount)}</span>
        `;
        item.classList.add(transaction.amount > 0 ?
            'income' : 'expense');
        transactionList.appendChild(item);
    });
}
```

This function first clears any current transactions in the list. Then, for each transaction in our `transactions` array, it creates a new list item (`li`) element, sets its inner HTML to display the transaction's description and amount, adds a class based on whether the transaction is an income or an expense, and then appends the item to our `transaction-list`.

4. Additional CSS for Income and Expense

Differentiate income and expense visually using colors:

```
``css
transaction-list .income {
    color: green;
}
transaction-list .expense {
    color: red;
}
``
```

5. Calling the `displayTransactions` Function

Whenever a new transaction is added or an existing one is modified, ensure you call the `displayTransactions()` function. This will ensure that the list is always updated with the latest data:

```
``javascript
// Sample code when adding a new transaction
function addTransaction(transaction) {
    transactions.push(transaction);
    displayTransactions();
}
``
```

Conclusion

With the above steps, we've successfully implemented the functionality to display transaction items in our Expense Tracker project. As users add or remove transactions, the list will update dynamically, providing a clear and concise view of their financial activities. In subsequent chapters, we will build upon this foundation by adding more features, such as calculating and

displaying balances, incomes, and expenses. Stay tuned!

Display Balance, Income & Expense

In the heart of our Expense Tracker, one of the most crucial functionalities is the ability to correctly display the user's balance, income, and expenses. This gives the user a clear overview of their financial status at a glance. In this chapter, we'll dive deep into the JavaScript logic and DOM manipulation techniques required to achieve this.

1. Setting the Stage

Before we write any code, it's essential to have the necessary HTML structure in place to hold our balance, income, and expense.

HTML Structure:

```
“html
<div class=“balance”>
    <h3>Your Balance</h3>
    <h2 id=“balance-amount”>$0.00</h2>
    <div class=“income-expense”>
        <div class=“plus”>
            <h3>Income</h3>
            <p id=“income-amount”>$0.00</p>
        </div>
        <div class=“minus”>
            <h3>Expense</h3>
            <p id=“expense-amount”>$0.00</p>
```

```
</div>
</div>
</div>
``
```

2. Initializing Our Variables

Within our script, let's initiate the required variables:

```
``javascript
let balance = 0;
let income = 0;
let expense = 0;
const balanceAmount =
document.getElementById('balance-amount');
const incomeAmount =
document.getElementById('income-amount');
const expenseAmount =
document.getElementById('expense-amount');
``
```

3. Calculation Function

Now, let's create a function that calculates the balance, income, and expense based on the transaction data.

```
``javascript
function calculateTotals(transactions) {
    income = transactions
        .filter(transaction => transaction.amount > 0)
        .reduce((acc, transaction) => acc +
    transaction.amount, 0);
    expense = transactions
        .filter(transaction => transaction.amount < 0)
``
```

```
    .reduce((acc, transaction) => acc +  
transaction.amount, 0) * -1; // We multiply by -1 to get a  
positive number  
  
    balance = income - expense;  
}  
“
```

Note: In the function above, we're using array methods `filter()` and `reduce()` to calculate our income, expense, and balance.

4. Displaying the Values

With our calculations done, it's time to display these values in our DOM.

```
“javascript  
function updateUI() {  
    balanceAmount.textContent =  
`$$\{balance.toFixed(2)\}`;  
    incomeAmount.textContent =  
`$$\{income.toFixed(2)\}`;  
    expenseAmount.textContent =  
`$$\{expense.toFixed(2)\}`;  
}  
“
```

5. Integrating with Local Storage

Since our project involves local storage, we want to make sure that whenever a new transaction is added or removed, we recalculate our values and update the UI.

```
“javascript  
function updateLocalStorage() {  
    localStorage.setItem('transactions',  
JSON.stringify(transactions));
```

```
    calculateTotals(transactions);
    updateUI();
}

```

```

## 6. Integrating with the Existing Transaction System

Assuming we have a mechanism to add or remove transactions, we should call the `updateLocalStorage` function right after a transaction is added or removed.

```
``javascript
function addTransaction(transaction) {
 transactions.push(transaction);
 updateLocalStorage();
}

function removeTransaction(id) {
 transactions = transactions.filter(transaction =>
transaction.id !== id);
 updateLocalStorage();
}

```


---



```

Conclusion

Our Expense Tracker now not only adds or removes transactions but also provides a clear overview of a user's financial status by displaying the balance, income, and expense in real-time. By integrating with local storage, we ensure that our data remains persistent across sessions. This project demonstrates the power of Vanilla JavaScript in building dynamic applications without relying on any libraries or frameworks. As we progress, we will continue adding more functionalities to

this project, making it a comprehensive tool for anyone wishing to track their expenses.

Exercise:

1. Add functionality to handle different currencies.
2. Enhance the UI to differentiate between positive and negative balances visually.
3. Extend the tracker to categorize expenses and represent them in a pie chart.

Add & Delete Transactions

In this chapter, we'll learn how to add and delete transactions within our Expense Tracker. We'll be interacting with the DOM to capture user input, and we'll utilize local storage to persist our transactions. Let's dive in.

1. The HTML Structure

Before adding or deleting transactions, we need a simple HTML structure to capture user input. For the sake of brevity, let's assume the following structure is in place:

```
``html
<div id="transaction-form">
  <input type="text" id="transaction-name"
placeholder="Enter transaction name...">
  <input type="number" id="transaction-amount"
placeholder="Enter amount...">
  <button id="add-transaction">Add
Transaction</button>
</div>
<ul id="transaction-list">
  <!-- Transactions will be dynamically added here -->
```

```
</ul>
```

```
“
```

2. Capturing User Input

Firstly, we need to capture what the user enters. This can be achieved using the DOM.

```
“`javascript
```

```
const transactionNameEl =  
document.getElementById('transaction-name');  
  
const transactionAmountEl =  
document.getElementById('transaction-amount');
```

```
“
```

3. Storing Transactions

We will store our transactions in an array and later save them to local storage.

```
“`javascript
```

```
let transactions = [];  
  
function addTransaction(e) {  
    e.preventDefault();  
  
    if (transactionNameEl.value.trim() === "" ||  
        transactionAmountEl.value.trim() === "") {  
  
        alert('Please enter a valid name and amount for the  
transaction.');//  
  
        return;  
    }  
  
    const newTransaction = {  
        id: generateID(),  
        name: transactionNameEl.value,  
        amount: parseFloat(transactionAmountEl.value)
```

```

};

transactions.push(newTransaction);
updateLocalStorage();
updateUI();
transactionNameEl.value = "";
transactionAmountEl.value = "";
}

function generateID() {
  return Math.floor(Math.random() * 100000000);
}
```

```

---

#### 4. Displaying Transactions

Once a transaction is added, we need to display it to the user.

```

``javascript
function updateUI() {
 const transactionListEl =
document.getElementById('transaction-list');
 transactionListEl.innerHTML = '';
 transactions.forEach(transaction => {
 const sign = transaction.amount < 0 ? '-' : '+';
 const item = document.createElement('li');
 item.className = transaction.amount < 0 ? 'minus' :
'plus';
 item.innerHTML = `
 ${transaction.name}
${sign}${Math.abs(transaction.amount)}
 <button class="delete-btn"
onclick="deleteTransaction(${transaction.id})">x</button
 `;
 transactionListEl.appendChild(item);
 });
}
```

```

```
>
  `;
  transactionListEl.appendChild(item);
});
}
``
```

5. Deleting Transactions

For deletion, we can utilize the unique ID we've given to each transaction.

```
``javascript
function deleteTransaction(id) {
  transactions = transactions.filter(transaction =>
transaction.id !== id);
  updateLocalStorage();
  updateUI();
}
``
```

6. Persisting to Local Storage

We want our transactions to remain available even after a page reload. For that, we'll utilize local storage.

```
``javascript
function updateLocalStorage() {
  localStorage.setItem('transactions',
JSON.stringify(transactions));
}
// Initially load transactions from local storage (if
available)
function initializeTransactions() {
```

```
const storedTransactions =  
localStorage.getItem('transactions');  
  
transactions = storedTransactions ?  
JSON.parse(storedTransactions) : [];  
  
updateUI();  
}  
  
initializeTransactions();  
``
```

7. Event Listeners

Finally, we attach an event listener to our add button:

```
``javascript  
document.getElementById('add-  
transaction').addEventListener('click', addTransaction);  
``
```

And voilà! With these functionalities in place, users can seamlessly add and delete transactions. The transactions are not only reflected in the UI but are also stored in the browser's local storage, ensuring no data is lost even if the user refreshes the page.

Remember, while the above code snippets provide a solid foundation for an expense tracker, there are various enhancements and additional features you can incorporate. As always, testing each functionality thoroughly ensures a smooth user experience.

Persist To Local Storage

One of the most common requirements for modern web applications is the ability to persist data across sessions, without necessarily sending data back to a server. This can provide users with a seamless experience, as data like settings, user preferences, or even user-generated content can be saved and then loaded on subsequent

visits. In our Expense Tracker project, we'll leverage the power of the Web Storage API, specifically the Local Storage, to achieve this.

What is Local Storage?

Local Storage is a web-based storage solution that allows websites to store key-value pairs in a web browser with no expiration time. It's perfect for saving user-specific data, and it's much larger in terms of storage capacity than cookies.

Benefits:

- Persistence: Unlike session storage, data stored in local storage does not expire with the session. This means even if you close the browser or tab, the data will still be there when you come back.
 - Capacity: You can store up to 5-10 MB of data depending on the browser. Far more than what cookies can hold.
 - Simplicity: The API is straightforward, with simple set, get, and remove methods.
-

Implementing Local Storage in the Expense Tracker

Storing Transactions:

To persist our expense transactions, we'll need to save them to local storage whenever a new transaction is added or an existing one is deleted.

```
``javascript
// Initial transactions array
let transactions = [];

function updateLocalStorage() {
    localStorage.setItem('transactions',
    JSON.stringify(transactions));
}
```

```
``
```

Adding a Transaction:

When you add a new transaction to your list:

```
``javascript
```

```
function addTransaction(transaction) {  
    transactions.push(transaction);  
    updateLocalStorage();  
}
```

```
``
```

This will add the transaction to the `transactions` array and then update local storage with the new array.

Deleting a Transaction:

Similarly, when deleting:

```
``javascript
```

```
function deleteTransaction(id) {  
    transactions = transactions.filter(transaction =>  
        transaction.id !== id);  
    updateLocalStorage();  
}
```

```
``
```

Here, we're filtering out the transaction with a specific ID and then updating our local storage.

Loading Transactions from Local Storage:

When the user visits the expense tracker, it's essential to check local storage and populate any saved transactions.

```
``javascript
```

```
function init() {  
    // Check for saved transactions in Local Storage
```

```
const savedTransactions =  
JSON.parse(localStorage.getItem('transactions'));  
  
if (savedTransactions) {  
  transactions = savedTransactions;  
  // Then, render these transactions to the DOM as  
  // needed  
}  
}  
  
init();  
``
```

Important Considerations:

- JSON Parsing and Stringifying: Local Storage only stores strings. So, when saving arrays or objects, you'll need to stringify them first using `JSON.stringify()`. When retrieving them, parse them back into a usable format with `JSON.parse()`.
- Limitations: While the 5-10 MB limit is generous for many applications, be aware of it. If there's a potential for your application to store a large amount of data, you may want to consider other options or strategies like IndexedDB or even server-side storage.
- Security: Local Storage is not designed for sensitive data. Since it's accessible via JavaScript, it's open to potential XSS attacks. Always ensure you validate and sanitize your data.

Conclusion:

Local Storage is a powerful tool in a web developer's arsenal, offering a simple and efficient way to enhance user experience through data persistence. In our Expense Tracker application, it provides an essential bridge between sessions, ensuring that our users never

lose track of their transactions. As with all tools, it's crucial to be aware of its limitations and use it judiciously.

Section 11:

Project 10 - Infinite Scroll Posts | Fetch, Async/Await, CSS Loader

Project Intro

Welcome to the tenth project of our journey, “Infinite Scroll Posts”. This project promises to be an engaging endeavor into the world of fetching data from APIs, utilizing asynchronous JavaScript operations, and enhancing the user experience with visually pleasing CSS loader animations. So, without further ado, let's dive into what this project entails.

1. Overview

In the modern era of the web, user experience stands paramount. An aspect of this user experience is the ability to fetch and display content without any noticeable interruptions. Gone are the days when users were accustomed to clicking ‘Next’ on pagination. Now, they expect content to automatically load as they scroll, popularly known as infinite scrolling. This feature can be seen on numerous modern websites, from social media platforms like Twitter and Facebook to news websites.

2. Objectives

By the end of this project, you will:

- Understand the basics and the mechanics behind the infinite scroll feature.
 - Fetch data from a public API using Fetch and Async/Await.
 - Display this fetched data seamlessly as the user scrolls down the page.
 - Enhance the user experience by adding a CSS loader animation to indicate data fetching.
-

3. Technologies & Concepts

- Fetch API: We'll be using the Fetch API to request and receive data. This powerful web API makes it easy to fetch resources across the network.
 - Async/Await: To handle asynchronous operations effectively, we'll leverage the power of Async/Await in JavaScript. This will ensure that our data fetching is smooth and we're not running into any unforeseen race conditions.
 - CSS Loader: A user must always be informed about the operations happening in the background. A CSS loader serves this purpose by providing a visual cue, indicating data is being fetched.
 - Infinite Scroll Mechanism: We will dive deep into the logic behind auto-loading content. By checking the user's scroll position and the height of the content, we can determine when to fetch more data.
-

4. What to Expect?

Here's a snapshot of what the final product will look like:

1. Initial Load: When the user first visits the site, a predefined number of posts will be displayed on the screen.
2. Scrolling: As the user scrolls down, just before reaching the bottom, new posts will automatically start loading, enhancing the UX.

3. Loader Animation: While new posts are fetched, a loader will be displayed at the bottom, indicating to the user that content is being loaded.
 4. Error Handling: In cases where the data cannot be fetched, perhaps due to network issues or API limits, the user will be notified appropriately.
-

5. Pre-requisites

Before diving into this project, it's beneficial if you're familiar with:

- Basic HTML/CSS for structure and styling.
 - Fundamentals of JavaScript, especially promises.
 - Basic understanding of how APIs work.
-

6. A Word Before We Begin

The “Infinite Scroll Posts” project will test your skills and provide real-world experience in creating a feature that's widely used in today's web applications. As we progress, remember that every challenge is a stepping stone to mastery. Stay curious, and keep coding!

In the upcoming chapters, we will start with the HTML structure, style our project using CSS, and then dive deep into the JavaScript logic that powers our infinite scrolling mechanism. Buckle up, and let's get started!

Project HTML

Welcome to Chapter 51, where we'll be delving into the HTML structure for our Infinite Scroll Posts project. This project is particularly exciting as it brings together the power of Fetch API, asynchronous functions, and some smooth CSS animations. The HTML structure forms the foundation on which our CSS and JavaScript will operate, so let's make sure we lay a strong foundation!

1. Project Overview

In the Infinite Scroll Posts project, we aim to create a web application where users can infinitely scroll through a series of posts, just like how it works on many social media platforms. As users scroll to the end of the page, new posts are fetched from an API and displayed seamlessly. In addition, a CSS loader will signal to the user that more content is being loaded.

2. Setting up the Document

Every HTML document starts with a basic structure. Let's begin with that:

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Infinite Scroll Posts</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <!-- Content goes here -->
    <script src="script.js"></script>
</body>
</html>
``
```

3. Main Content Area

The core content of our project is the posts that we'll fetch and display. Let's create a container for these posts:

```
``html
<div class="posts-container">
    <!-- Individual posts will be appended here -->
</div>
``
```

4. Loader

As we're implementing infinite scrolling, we need a loader to inform users when new posts are being fetched:

```
``html
<div class="loader">
    Loading...
</div>
``
```

(Note: The actual visual representation of the loader will be controlled using CSS. In our JavaScript, we will programmatically show or hide this loader based on whether we're fetching data.)

5. Error Message

In case there's an issue fetching posts, it's good UX to have a designated area to display an error message:

```
``html
<div class="error-message">
    <!-- Error messages will be inserted here -->
</div>
``
```

6. Adding Metadata

It's good practice to include meta tags to improve SEO (Search Engine Optimization) and ensure a good display when shared on platforms like Facebook or Twitter:

```
``html
<meta name="description" content="Infinite Scroll Posts:
Dive into endless content from our API.">
<meta property="og:title" content="Infinite Scroll Posts">
<meta property="og:description" content="Experience
the magic of infinite scrolling with posts fetched in real-
time from our API.">
<meta property="og:image"
content="path_to_thumbnail_image.jpg">
``
```

Replace `path_to_thumbnail_image.jpg` with the path to your chosen thumbnail image for sharing on social platforms.

7. Closing Thoughts

The HTML structure we've laid out here forms the base of our Infinite Scroll Posts project. While it might look simple, remember that the magic happens when our CSS and JavaScript come into play, making this structure come to life.

In the following chapters, we'll be styling this structure using CSS and then adding the functionality with JavaScript. Stick around to see how our simple HTML transforms into a dynamic infinite scrolling experience!

Project CSS & Loader Animation

Welcome to Chapter 59! Here, we'll be discussing the CSS design and the creation of a loader animation for our Infinite Scroll Posts project. By the end of this chapter, you should have a visually appealing layout and a smooth loading animation that will enhance the user experience, especially when fetching new posts.

Section 1: Basic CSS Setup

Before diving into the specifics of our loader animation, let's ensure that we set up our basic CSS foundation.

```
``css
body {
    font-family: 'Arial', sans-serif;
    background-color: f9f9f9;
    color: 333;
    line-height: 1.6;
}

.container {
    width: 80%;
    margin: auto;
    overflow: hidden;
}

.post {
    border: 1px solid ccc;
    padding: 20px;
    margin: 20px 0;
    border-radius: 5px;
    background-color: fff;
    box-shadow: 2px 2px 12px rgba(0,0,0,0.1);
}
```

“

In this initial setup:

- We're setting a general font, background color, text color, and line-height for the entire body.
 - The ` .container` class centers our content and keeps everything tidy.
 - Individual ` .post` sections are clearly defined with subtle borders, padding, shadows, and rounded edges for a modern appearance.
-

Section 2: Designing the Loader Animation

For our infinite scroll, it's important to provide user feedback. The loader animation is a visual indicator that posts are currently being fetched.

Step 1: Basic Loader Structure

Our loader will be a simple spinning circle. Begin by adding this HTML structure wherever you intend to place your loader:

```
``html
<div class="loader"></div>
``
```

Step 2: Styling the Loader

Now, apply the following CSS to design our loader:

```
``css
.loader {
    border: 16px solid f3f3f3; /* Light grey */
    border-top: 16px solid 3498db; /* Blue */
    border-radius: 50%; /* To make it a circle */
    width: 120px;
    height: 120px;
    animation: spin 1s linear infinite;
}
```

```
}
```

```
"
```

Step 3: Creating the Spin Animation

Now, the magical part. Let's create the `spin` animation:

```
"`css
```

```
@keyframes spin {  
    0% { transform: rotate(0deg); }  
    100% { transform: rotate(360deg); }  
}
```

```
"
```

With this keyframes animation, our loader will continuously spin, providing a visual cue to users that content is loading.

Section 3: Positioning the Loader

Our loader should be centrally placed on the page, especially when posts are being fetched. Apply the following styles to ensure its position remains consistent:

```
"`css
```

```
.loader {  
    position: fixed;  
    left: 50%;  
    top: 50%;  
    transform: translate(-50%, -50%);  
    z-index: 1000;  
}
```

```
"
```

Section 4: Showing and Hiding the Loader

With JavaScript, you can easily control the visibility of the loader. While fetching posts, set its style to `display: block`, and once done, set it to `display: none`.

```
“javascript
// Assuming you've selected your loader with a query
const loader = document.querySelector('.loader');

// To show the loader
loader.style.display = 'block';

// To hide the loader
loader.style.display = 'none';
“
```

This way, our loader will only be visible during the fetching process, ensuring users aren't unnecessarily distracted.

Conclusion

CSS plays a vital role in enhancing user experience, especially for web applications like our Infinite Scroll Posts project. With a visually appealing design and a smooth loader animation, we're making the user experience delightful and intuitive.

In the next chapter, we will focus on fetching the initial posts and populating our page with content. This will be where the real magic happens, as we combine our CSS designs with dynamic JavaScript functionalities. Stay tuned!

Get Initial Posts From API

In our “Infinite Scroll Posts” project, the first step before implementing the infinite scroll feature is to fetch initial posts from an API. In this chapter, we will delve deep into how to get posts from an external API using the `Fetch` API and `Async/Await`. Before starting, make

sure you have set up your project structure correctly as described in the previous chapters.

Why Fetch API with Async/Await?

JavaScript's Fetch API provides a simpler way of making network requests to the server. When combined with Async/Await, Fetch becomes even more powerful, allowing us to write asynchronous code that looks and behaves like synchronous code, making it easier to read and maintain.

Setting Up The API Endpoint

For this tutorial, let's use the `jsonplaceholder` free fake online REST API, which provides a set of posts that we can retrieve:

API Endpoint:

`<https://jsonplaceholder.typicode.com/posts>`

Writing the Fetch Function with Async/Await

```
“javascript
async function getInitialPosts() {
    try {
        const response = await
fetch('https://jsonplaceholder.typicode.com/posts?
_limit=5');
        const data = await response.json();
        return data;
    } catch (error) {
        console.error("There was an error fetching the
posts", error);
    }
}
```

```
“
```

Understanding the Function

- We declare the function `getInitialPosts` as `async`. This ensures that the function returns a promise.
- Inside the function, we use a `try...catch` block to handle any potential errors.
- We call the `fetch` function with the API endpoint. To ensure that we don't get too many posts initially, we use a query parameter `_limit=5` to limit the number of posts to 5.
- The `await` keyword is used to pause the execution of the function until the promise settles and returns its result. This makes our asynchronous code appear synchronous.
- Once the data is fetched, we convert it to JSON using the `response.json()` method.
- If everything goes smoothly, the function will return the fetched posts as a JSON array.

Displaying the Posts

Once we've fetched the posts, we need to display them on our page. Let's add another function to handle this.

```
“javascript
```

```
function displayPosts(posts) {  
    const postContainer =  
        document.getElementById('posts');  
    posts.forEach(post => {  
        const postElement =  
            document.createElement('div');  
        postElement.classList.add('post');  
        postElement.innerHTML = `
```

```
<h2 class="post-title">${post.title}</h2>
<p class="post-body">${post.body}</p>
';
postContainer.appendChild(postElement);
});

}

``
```

To use this function:

```
``javascript
getInitialPosts().then(posts => {
  displayPosts(posts);
});
``
```

Conclusion

In this chapter, we covered how to use the Fetch API combined with Async/Await to retrieve initial posts from an API endpoint. We also looked at how to display these posts on our webpage. In the next chapter, we will add the infinite scrolling feature, which will load more posts as the user scrolls down the page.

Remember, while `jsonplaceholder` is great for learning and testing, in a real-world application, you'd be working with your own backend or a third-party service that provides the content you need. Always ensure you handle errors gracefully to enhance the user experience.

Add Infinite Scrolling

Welcome to Chapter 61! Here, we'll be implementing one of the most sought-after features on modern websites: Infinite Scrolling. This technique enhances

user experience by continuously loading content as users scroll, removing the need for pagination buttons. To achieve this, we'll harness the power of Fetch and Async/Await in conjunction with some CSS magic. Let's dive in!

1. Understanding Infinite Scrolling

Infinite scrolling is a web design technique that loads content continuously as the user scrolls down the page, eliminating the need for pagination. It's widely used in social media platforms, blogs, and news websites to improve user engagement. The principle is simple: as the user approaches the bottom of the content, more content is fetched and appended.

2. Setting up the Environment

Before implementing infinite scrolling, ensure you have:

- An API or data source from which you can fetch posts or data.
 - A container in your HTML to display the fetched data.
 - Basic styling for the loader animation (Refer to Chapter 59 for the CSS & Loader Animation).
-

3. Listening for Scroll Events

To detect when the user is near the bottom of the page, we'll use the `scroll` event.

```
``javascript
window.addEventListener('scroll', () => {
  if (window.innerHeight + window.scrollY >=
    document.documentElement.scrollHeight - 10) {
    loadMorePosts();
  }
});
```

“

The condition inside the event checks if the combined height of the viewport and the scroll offset is greater than or equal to the height of the entire document minus a small threshold (10 pixels in this case).

4. Fetching More Posts

The `loadMorePosts` function will handle the fetching of more posts.

``javascript

```
async function loadMorePosts() {  
    // Display the loader animation  
    showLoader();  
    // Delay to simulate fetching data  
    await setTimeout(() => {}, 1000);  
    // Fetch the next set of posts  
    const response = await  
    fetch('YOUR_API_ENDPOINT_HERE');  
    const data = await response.json();  
    // Render the fetched posts  
    displayPosts(data);  
    // Hide the loader animation  
    hideLoader();  
}
```

“

Here, `showLoader()` and `hideLoader()` are functions to handle displaying and hiding the loader animation, respectively. Ensure to replace `‘YOUR_API_ENDPOINT_HERE’` with your actual API endpoint.

5. Appending the Posts

Within the `displayPosts` function, iterate over the fetched data and append each post to your container.

```
``javascript
function displayPosts(posts) {
    posts.forEach(post => {
        const postElement =
document.createElement('div');
        postElement.classList.add('post');
        postElement.innerHTML = `
            <h2>${post.title}</h2>
            <p>${post.body}</p>
        `;
        document.querySelector('.posts-
container').appendChild(postElement);
    });
}
``
```

Ensure you have a `'.posts-container` or similar in your HTML to append these posts to.

6. Handling Edge Cases

For a seamless user experience:

- Handle cases where there's no more content to fetch. Inform the user accordingly.
 - Ensure the loader doesn't get stuck if there's an error during fetching.
 - Adjust the threshold (10 pixels in our case) depending on your content's structure and the desired user experience.
-

Conclusion

Infinite scrolling is a powerful technique for improving user engagement on content-rich websites. By utilizing modern JavaScript features like Fetch and Async/Await, we can easily implement this functionality in our web projects. Always remember to handle edge cases and provide feedback to users for a seamless browsing experience.

Note: This chapter assumes you have a basic understanding of Fetch API, Async/Await, and CSS animations. If not, kindly revisit the previous chapters for a thorough understanding. The code provided is a basic implementation and might require adjustments based on the specifics of your project and API.

Filter Fetched Posts

Welcome back to the “Infinite Scroll Posts” project. Having already fetched posts and implemented infinite scrolling, our next goal is to introduce a functionality that allows users to filter through the fetched posts. This will enhance user experience by enabling them to find specific posts quickly.

Objective: Implement a filtering functionality that allows users to search through the fetched posts based on titles and content.

Prerequisites:

1. A working knowledge of JavaScript, particularly ES6+ features such as Async/Await.
 2. Familiarity with the DOM (Document Object Model) and manipulating it with JavaScript.
 3. Previous chapters in this project where we set up fetching posts and infinite scrolling.
-

Step 1: Setting up the Filter Input Field

We need an input field where users can type search queries:

```
``html
<input type="text" id="filter" placeholder="Filter posts...">
``
```

Step 2: JavaScript Implementation

a. Accessing the Input Element:

Access the input element using the DOM:

```
``javascript
const filterInput = document.getElementById('filter');
``
```

b. Event Listener:

We'll attach an event listener to the input element to detect when the user types:

```
``javascript
filterInput.addEventListener('input', filterPosts);
``
```

c. Filtering Function:

This function is the core of our filtering feature:

```
``javascript
function filterPosts(e) {
    const searchTerm = e.target.value.toUpperCase(); // Convert input to uppercase for non-case sensitive search
    const posts = document.querySelectorAll('.post'); // Assume each post has a class of 'post'
    posts.forEach(post => {
```

```

        const postTitle = post.querySelector('.post-
title').innerText.toUpperCase();

        const postContent = post.querySelector('.post-
content').innerText.toUpperCase();

        // Check if search term exists in post title or
        content

        if (postTitle.indexOf(searchTerm) > -1 ||

postContent.indexOf(searchTerm) > -1) {

            post.style.display = 'block';

        } else {

            post.style.display = 'none';

        }

    });

}

```

```

### Explanation:

- We're using the `input` event, which fires every time the `value` of the input field changes.
- Within the `filterPosts` function:
  - Convert the search term to uppercase to ensure our search is not case-sensitive.
  - We select all posts. For this example, I've assumed each post has a class of 'post'.
  - For each post, we check if the search term exists in its title or content. If it does, we display the post; otherwise, we hide it.

### Step 3: Enhancements

- a. Adding a Delay: Instant filtering can be jarring, especially with larger datasets. Consider adding a delay using `setTimeout`.

- b. Spinner or Loader: If filtering takes time, showing a spinner can enhance user experience. You can reuse the CSS loader implemented in previous chapters.
- c. Highlighting Matched Text: Enhance user experience by highlighting the matched text within the post. This can be achieved using Regex and manipulating the innerHTML property of post elements.

---

#### Conclusion:

Filtering is a powerful feature that greatly enhances user experience, allowing users to quickly and efficiently find the information they're looking for. With the implementation above, users of your infinite scroll post project can now search through posts with ease.

## Section 12: Project 11 - Speech Text Reader | Speech Synthesis

### Project Intro

As the digital landscape evolves, the ways in which we interact with our devices are also rapidly changing. While touch and type are the primary modes of interaction, voice recognition and synthesis are quickly making their mark in the world of technology. The ability of machines to understand and respond to our voice not only offers convenience but also bridges the gap for users with disabilities.

In this project, we will embark on a journey to develop a “Speech Text Reader” application. Leveraging the Web Speech API, particularly the Speech Synthesis interface, our application will be capable of converting text input into audible speech. Whether you’re thinking about a tool

that aids visually impaired users, or simply an application that can read out loud a bedtime story, the possibilities are endless.

---

## Objectives

By the end of this project, you will be able to:

1. Understand the fundamentals of the Web Speech API and its role in modern web applications.
  2. Implement the Speech Synthesis interface to convert textual data into speech.
  3. Customize voice selections, pitch, and rate of speech.
  4. Design a user-friendly interface that facilitates easy text input and speech output.
- 

## Why Speech Synthesis?

Speech synthesis, commonly known as text-to-speech (TTS), has numerous applications:

- Accessibility: It aids users with visual impairments, dyslexia, or other conditions that make reading on screen challenging.
  - Multitasking: Users can listen to content while engaging in other activities, such as driving or cooking.
  - Language Learning: It provides pronunciation guidance for people learning new languages.
  - Entertainment: For applications like reading stories or converting written content into podcasts.
- 

## How Does it Work?

The Web Speech API provides a bridge to the text-to-speech engines, allowing developers to control voice output from their web apps. With the Speech Synthesis interface, we'll have a range of options to adjust the voice (male, female, etc.), the pitch, rate, and even select different languages.

## Scope of the Project

Our Speech Text Reader will consist of:

- A simple and clean User Interface (UI) for text input.
- Options to select different voices.
- Sliders/controls to adjust pitch and rate.
- A play button to initiate the speech synthesis.
- Visual feedback indicating when the text is being read out.

## What to Expect in the Following Chapters

As we progress through the chapters, we will:

1. Set up the HTML framework for our application, focusing on the user interface.
2. Dive deep into the CSS, ensuring our application is not just functional but also aesthetically pleasing.
3. Implement the JavaScript, where the magic happens, integrating the Web Speech API, capturing user inputs, and converting them into voice.

## Pre-requisites

While this project is tailored for learners who have some experience with HTML, CSS, and JS, it's designed to be approachable for beginners as well. If you're completely new to these technologies, it might be beneficial to revisit the introductory chapters or courses as mentioned in the book description.

## Let's Get Started!

Excited? We are too! The ability to convert text into speech is a remarkable one, and by the end of this project, you'll have a fully functional Speech Text Reader. So, without further ado, let's dive right in!

# HTML & Output Speech Boxes

In Project 11, we are venturing into the realm of web speech synthesis, giving our web applications the ability to “speak” to our users. The Speech Synthesis API provides a bridge between web applications and system-level speech synthesis software. This chapter will walk you through creating the HTML structure for our Speech Text Reader, focusing on outputting speech boxes that will be the foundation for our reader.

---

## 1. The Structure of Our Speech Reader

For our Speech Text Reader, we’ll have a collection of preset phrases or sentences that users can click on to hear them spoken aloud. Each phrase will be contained within its own box, which we’ll refer to as a “speech box”.

---

## 2. Setting Up the Base HTML Structure

Let’s start by setting up the foundational HTML structure.

```
“html
<!DOCTYPE html>
<html lang=“en”>
<head>
 <meta charset=“UTF-8”>
 <meta name=“viewport” content=“width=device-width,
initial-scale=1.0”>
 <title>Speech Text Reader</title>
 <link rel=“stylesheet” href=“path_to_your_css”>
</head>
<body>
```

```
<main id="main">
 <h1>Speech Text Reader</h1>
 <div class="speech-boxes">
 <!-- Speech boxes will be added here -->
 </div>
</main>
<script src="path_to_your_js"></script>
</body>
</html>
"
```

---

### 3. Creating a Speech Box

Each speech box will contain a phrase or sentence that can be clicked to be read aloud. Here's what the HTML structure of a single speech box might look like:

```
"`html
<div class="speech-box" data-text="Hello, world!">
 <p>Hello, world!</p>
</div>
"
"
```

The `data-text` attribute holds the text that will be read aloud. We'll use JavaScript to access this attribute and then speak the text using the Speech Synthesis API.

---

### 4. Adding More Speech Boxes

For our example, let's add a few more boxes with different phrases:

```
"`html
<div class="speech-boxes">
 <div class="speech-box" data-text="Hello, world!">
```

```
<p>Hello, world!</p>
</div>

<div class="speech-box" data-text="How are you
today?">
 <p>How are you today?</p>
 </div>

 <div class="speech-box" data-text="It's a sunny day
outside.">
 <p>It's a sunny day outside.</p>
 </div>
 </div>
``
```

Each speech box contains a unique phrase. By clicking on any of these boxes, the user will be able to hear the corresponding text.

---

## 5. Enhancing User Experience

To enhance the user experience, consider adding some visuals or icons that give a hint to the user that these boxes are clickable and will produce audio output. For instance, you could add a small speaker or audio icon inside each box.

---

## Summary

This chapter introduced the basic HTML structure required for our Speech Text Reader application. We designed speech boxes to house preset phrases that users can click on to hear them spoken aloud. In the following chapters, we will delve into the Speech Synthesis API and the corresponding JavaScript to bring our speech boxes to life.

The visual appearance of our speech boxes and the overall layout can be improved and made interactive with

CSS, which we will cover in the next chapter.

# Project CSS

Welcome to the CSS styling part of our Speech Text Reader project! Having completed the HTML structure, we now need to enhance its look and feel to make it both user-friendly and attractive. In this chapter, we'll walk through every CSS property and style we'll apply to our Speech Text Reader to make it come to life.

---

## 1. Setting Up the Base Styles

First, let's reset some default browser styles and establish a base for our project.

```
``css
* {
 margin: 0;
 padding: 0;
 box-sizing: border-box;
 font-family: 'Arial', sans-serif;
}

body {
 background-color: f4f4f4;
 color: 333;
 font-size: 16px;
 line-height: 1.5;
}
``
```

---

## 2. Main Container

Our main container will center everything on the screen and give a gentle shadow for depth.

```
``css
.container {
 max-width: 800px;
 margin: 50px auto;
 padding: 20px;
 background: fff;
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}
``
```

---

### 3. Heading

The heading style will be bold, centered, and have a bit more space on the bottom margin to separate it from the content below.

```
``css
h1 {
 text-align: center;
 font-size: 2em;
 margin-bottom: 20px;
}
``
```

---

### 4. Speech Box Styling

These boxes will contain predefined text that can be read out loud by our application.

```
``css
.speech-box {
 display: flex;
```

```
 align-items: center;
 justify-content: space-between;
 padding: 15px;
 border: 1px solid ccc;
 margin-bottom: 10px;
 cursor: pointer;
 transition: background 0.3s ease;
}
.speech-box:hover {
 background: f9f9f9;
}
``
```

---

## 5. Custom Text Box

This is where the user can input custom text for the application to read.

```
``css
.custom-text {
 margin-top: 20px;
}
.custom-text textarea {
 width: 100%;
 padding: 10px;
 border: 1px solid ccc;
 height: 100px;
 resize: none;
}
``
```

## 6. Buttons

Our buttons will initiate the speech function. We want them to be easily identifiable and clickable.

```
``css
.btn {
 display: inline-block;
 background: 007BFF;
 color: fff;
 padding: 10px 20px;
 margin-top: 20px;
 border: none;
 cursor: pointer;
 transition: background 0.3s ease;
}
.btn:hover {
 background: 0056b3;
}
.btn:disabled {
 background: ccc;
 cursor: not-allowed;
}
``
```

---

## 7. Voice Selection Dropdown

This dropdown will allow the user to select different voice options.

```
``css
.voice-selection {
 margin-top: 20px;
```

```
}

.voice-selection select {
 width: 100%;
 padding: 10px;
 border: 1px solid ccc;
}

``
```

---

## Conclusion

With the styles set up, our Speech Text Reader should look polished and be user-friendly. These styles enhance the usability of the application and create a pleasant experience for the user. In the next chapter, we'll dive into JavaScript to integrate the Web Speech API, allowing the user to transform text into spoken words.

# Get Speech Voices

In the world of web development, providing a holistic user experience involves not only what the user sees but also what they hear. The Web Speech API's `SpeechSynthesis` interface offers the capability to vocalize text content. One of the key elements of this interface is the ability to select different voice options to suit the content's needs. In this chapter, we'll explore how to fetch and use the available voice options to enrich the Speech Text Reader project.

---

## Understanding `SpeechSynthesis`

Before diving into retrieving available voices, it's essential to grasp the basics of `'SpeechSynthesis'`. It's the primary interface for controlling text-to-speech on a web page. This interface possesses a method called `'getVoices()'`, which returns an array of

`SpeechSynthesisVoice` objects representing all the voices available.

---

## Fetching Available Voices

1. Initialize the Process: Before retrieving voices, we'll need a reference to the SpeechSynthesis interface:

```
``javascript
```

```
const synth = window.speechSynthesis;
```

```
``
```

2. Get and List Voices: With the initialized `synth` object, use the `getVoices()` method:

```
``javascript
```

```
let voices = [];
```

```
function populateVoiceList() {
```

```
 voices = synth.getVoices();
```

```
 // Code to display these voices will go here
```

```
}
```

```
``
```

Remember that the list of available voices might not be immediately available when the page loads. To handle this, we should listen to the `voiceschanged` event:

```
``javascript
```

```
synth.onvoiceschanged = populateVoiceList;
```

```
``
```

---

## Displaying the Voices in a Dropdown

To let users choose a voice, it's common practice to display them in a dropdown list:

1. HTML Structure:

```
``html
```

```
<select id="voiceSelect"></select>
```

```
"
```

## 2. Populate the Dropdown:

Back in the `populateVoiceList` function, you can now loop through the `voices` array and create an option for each voice:

```
"javascript
```

```
const voiceSelect =
document.getElementById('voiceSelect');
voices.forEach(voice => {
 const option = document.createElement('option');
 option.textContent = `${voice.name} (${voice.lang})`;
 // Set necessary attributes
 option.setAttribute('data-lang', voice.lang);
 option.setAttribute('data-name', voice.name);
 voiceSelect.appendChild(option);
});
```

```
"
```

---

## Using a Selected Voice

To actually use one of the voices, you'll need to set it when you're synthesizing speech:

### 1. Synthesize Speech:

Let's say you want to speak the following message:

```
"javascript
```

```
const utterThis = new SpeechSynthesisUtterance('Hello,
how are you?');
```

```
"
```

### 2. Set the Voice:

Before you instruct `synth` to speak the message, set the selected voice:

```
``javascript
const selectedVoiceName =
voiceSelect.selectedOptions[0].getAttribute('data-name');
utterThis.voice = voices.find(voice => voice.name ===
selectedVoiceName);
``
```

### 3. Speak!:

Finally, use the `speak` method:

```
``javascript
synth.speak(utterThis);
``
```

---

## Conclusion

Integrating dynamic voice selection into your Speech Text Reader enhances customization, catering to a global audience with varying linguistic preferences. With just a few lines of code, the Web Speech API's SpeechSynthesis interface grants your web applications a powerful tool, breathing life into static text. As you progress, consider exploring other features like adjusting pitch, rate, and volume to further tailor the user's auditory experience.

# Speech Buttons

In this chapter, we'll explore one of the core components of our Speech Text Reader project: the Speech Buttons. These buttons will serve as the trigger for our application to start converting the selected or entered text into audible speech. Harnessing the power of the Web Speech API, particularly the Speech Synthesis interface, we will give our application a voice.

## Prerequisites

Before diving into this chapter, ensure you've successfully set up the basic structure of the project as covered in the previous chapters.

## Creating the Speech Buttons

### 1. HTML Structure:

First, let's lay out the basic structure for our buttons in our HTML:

```
``html
<div id="buttons">
 <button id="start">Start Speaking</button>
 <button id="stop">Stop Speaking</button>
</div>
``
```

Here, we have two buttons:

- `Start Speaking`: To initiate the speech.
- `Stop Speaking`: To halt any ongoing speech.

### 2. Basic Styling:

For our buttons to be visually appealing and user-friendly, let's apply some CSS:

```
``css
buttons {
 display: flex;
 justify-content: space-around;
 margin-top: 20px;
}
button {
 padding: 10px 20px;
```

```
 font-size: 16px;
 cursor: pointer;
 border: none;
 background-color: #3498db;
 color: fff;
 border-radius: 5px;
}

button:hover {
 background-color: #2980b9;
}
“
```

## JavaScript Implementation

Now, let's harness the power of the Web Speech API:

### 1. Selecting our Elements:

Using JavaScript, we first need to select our button elements:

```
```javascript  
const startBtn = document.getElementById('start');  
const stopBtn = document.getElementById('stop');  
```
```

### 2. Initializing Speech Synthesis:

Next, we initialize our SpeechSynthesis object:

```
```javascript  
const synth = window.speechSynthesis;  
```
```

### 3. Start Speaking Button:

The Start Speaking button will take the provided text and convert it into speech. Let's see how this is done:

```
``javascript
startBtn.addEventListener('click', () => {
 if (synth.speaking) {
 console.error('Speech synthesis is already
speaking.');
 return;
 }

 let textInput = document.getElementById('text-
input').value; // Assuming 'text-input' is the ID of the
textarea or input where users type/paste their text

 if (textInput !== '') {
 let utterance = new
SpeechSynthesisUtterance(textInput);

 // Optional: Set voice, pitch, and rate here
 // utterance.voice = synth.getVoices()[0];
 // utterance.pitch = 1;
 // utterance.rate = 1;
 synth.speak(utterance);
 }
});

``
```

#### 4. Stop Speaking Button:

The Stop Speaking button will halt any ongoing speech synthesis:

```
``javascript
stopBtn.addEventListener('click', () => {
 if (synth.speaking) {
 synth.cancel();
 }
});
```

“

## Testing the Functionality

Once you've added the above code snippets, you should test the application in a browser that supports the Web Speech API. Type or paste some text into your designated input area and click the “Start Speaking” button. You should hear the text being read aloud. Clicking the “Stop Speaking” button should immediately halt the speech.

## Conclusion

The Speech Buttons play a pivotal role in our Speech Text Reader application, allowing users to control when text-to-speech conversion begins and ends. Leveraging the capabilities of the Web Speech API, we've managed to introduce a powerful feature with just a few lines of code. In the upcoming chapters, we'll explore more features and functionalities to make our application more robust and user-friendly.

# Change Voice & Custom Text Box

In this chapter, we will extend the functionalities of our Speech Text Reader by implementing two key features:

1. Changing the voice for our text-to-speech reader.
2. Allowing users to input custom text to be read out loud.

These features will make our application versatile, giving users more control over the voice output and the content being read.

## Step 1: Setting Up the HTML for Voice Selection and Custom Text

Before we dive into the JavaScript functionalities, let's set up our HTML to include a dropdown list for voice selection and a text area for custom text input.

index.html

```
``html
<div class="voice-selection">
 <label for="voiceList">Choose Voice:</label>
 <select id="voiceList"></select>
</div>

<div class="custom-text">
 <label for="customText">Enter Custom Text:</label>
 <textarea id="customText" rows="5"
placeholder="Type your custom text here..."></textarea>
 <button id="readCustomText">Read Custom
Text</button>
</div>
``
```

---

## Step 2: Fetching Available Voices

Let's fetch the available voices from the SpeechSynthesis API and populate our dropdown list.

app.js

```
``javascript
const voiceList = document.getElementById('voiceList');
let voices = [];
function populateVoiceList() {
 voices = speechSynthesis.getVoices();
 voices.forEach(voice => {
```

```
const option = document.createElement('option');
option.textContent = voice.name;
option.setAttribute('data-lang', voice.lang);
option.setAttribute('data-name', voice.name);
voiceList.appendChild(option);

});

}

// This event is triggered when voice list changes
speechSynthesis.onvoiceschanged = populateVoiceList;
"
```

---

### Step 3: Changing the Voice

Once the voices are loaded into our dropdown, we want the ability to change the voice based on the user's selection.

app.js (Continuation)

```
"`javascript
let selectedVoice;
voiceList.addEventListener('change', (e) => {
 selectedVoice = voices.find(voice => voice.name ===
e.target.value);
});
```

---

### Step 4: Implementing the Custom Text Box

For our custom text feature, we need to allow users to type in a text area, and then read the content out loud using the selected voice.

app.js (Continuation)

```
"`javascript
```

```
const customText =
document.getElementById('customText');

const readCustomTextButton =
document.getElementById('readCustomText');

readCustomTextButton.addEventListener('click', () => {
 let utterThis = new
SpeechSynthesisUtterance(customText.value);
 if (selectedVoice) {
 utterThis.voice = selectedVoice;
 }
 speechSynthesis.speak(utterThis);
 customText.value = ""; // Clear the text area after
reading
});
“
```

---

## Step 5: Styling the Interface

To keep our user interface intuitive and neat, let's add some styling.

styles.css

```
``css
.voice-selection, .custom-text {
 margin: 20px 0;
}

label {
 display: block;
 margin-bottom: 10px;
 font-weight: bold;
}

select, textarea {
```

```
width: 100%;
padding: 10px;
border: 1px solid ccc;
border-radius: 4px;
}

button {
 display: block;
 margin-top: 10px;
 padding: 10px 20px;
 background-color: 333;
 color: fff;
 border: none;
 border-radius: 4px;
 cursor: pointer;
}

button:hover {
 background-color: 555;
}
“
```

---

## Conclusion

With the steps outlined above, we've successfully added functionalities to change the voice of our Speech Text Reader and allow users to input custom text for the application to read out loud. This not only makes our application more versatile but also enhances the user experience by providing more control over the content and its presentation.

Remember, accessibility tools like these can make a difference in how users interact with web applications.

Always ensure that you test these tools with a variety of users to guarantee that they cater to different needs.

## Section 13: Project 12 - Relaxer App | CSS Animations, setTimeout

### Project Intro

Welcome to Project 12, the Relaxer App. In our fast-paced world, taking a few moments to relax and breathe is not just essential for mental health, but also for boosting productivity. With this in mind, the Relaxer App serves as a digital companion that encourages its users to pause, breathe, and reconnect with the present moment.

---

#### Objective of the Relaxer App:

The primary purpose of the Relaxer App is to guide the user through a series of breathing exercises, helping them achieve a state of calm and focus. As developers, it provides us an opportunity to delve into an interesting blend of CSS animations and JavaScript functions, especially the setTimeout method.

---

#### What Will We Build?

1. Visual Breathing Guide: A user-friendly UI that will visually show the user when to breathe in, hold their breath, and breathe out.
2. Animation: Using CSS, we'll create a smooth and calming animation that coincides with the breathing instructions, allowing the user to follow along visually.

3. JavaScript Integration: Our JS code will not only control the animation triggers but also guide the user on how long they should be performing each part of the breathing exercise.

---

#### Features of the Relaxed App:

1. Guided Breathing Instructions: Clear on-screen prompts that tell the user exactly when to breathe in, hold, and breathe out.
  2. Adaptive Cycle: The app will continually run the breathing cycle, allowing users to use it for as long as they need.
  3. Responsive Design: Our app will be adaptable to different devices ensuring everyone can access and benefit from it.
  4. Customizable Settings: While this is a basic version, developers can later add features where users can set their breathing intervals or choose different relaxation modes.
- 

#### Why the Relaxed App?

- Holistic Learning: This project offers a well-rounded learning experience, from creating visually appealing designs to understanding and implementing JavaScript's asynchronous functions.
- Real-world Application: Meditation and relaxation apps are increasingly popular. This project can serve as a foundation for those looking to delve deeper into this genre of apps.
- Challenging Yet Achievable: For those who've journeyed with us through the previous 11 projects, the Relaxed App provides just the right amount of challenge while ensuring that it remains achievable and enjoyable.

In conclusion, by the end of this project, you will not only have a functional Relaxed App but also a deeper

understanding of how CSS animations can be effectively combined with JavaScript methods to create a synchronous user experience. So, let's dive in and start building our path to relaxation and coding mastery!

## Creating The Large Circle

In the Relaxed App, the main visual component is a large circle which will represent our breathing visual guide. This circle will expand and contract, simulating the breathing process. Before animating it, we first need to create it. In this chapter, we'll walk you through the process of creating this essential component using HTML and CSS.

---

### Step 1: Setting up the HTML Structure

Let's start by creating the basic structure for our circle in the HTML.

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width,
initial-scale=1.0">
 <title>Relaxed App</title>
 <link rel="stylesheet" href="styles.css">
</head>
<body>
 <div class="container">
 <div class="circle"></div>
 </div>
```

```
</body>
</html>
``
```

Here, we've wrapped the circle inside a container. This will help in positioning and animating it later on.

---

## Step 2: Styling the Circle using CSS

For our circle to look visually appealing and give the user a sense of relaxation, we'll opt for a calm, pastel color. The size of the circle should be significant enough to be the primary focus, but not overwhelming.

In your `styles.css`:

```
``css
body, html {
 height: 100%;
 margin: 0;
 display: flex;
 justify-content: center;
 align-items: center;
 background-color: 282c34; /* A dark background for
contrast */
}
.container {
 position: relative;
 width: 300px;
 height: 300px;
}
.circle {
 position: absolute;
 top: 50%;
```

```
left: 50%;
transform: translate(-50%, -50%);
width: 100%;
height: 100%;
border-radius: 50%; /* Makes it a perfect circle */
background-color: 89CFF0; /* A soothing blue color */
box-shadow: 0px 0px 15px rgba(0,0,0,0.2); /* A subtle
shadow for depth */
}
“
```

In the above CSS, we've:

- Centered the ` `.container` using flex properties on the `body`.
- Used the `position: absolute` and `transform` properties for the ` `.circle` to ensure it's perfectly centered within the ` `.container` .
- Used the `border-radius` property to turn our square div into a circle.
- Chose a relaxing blue shade for the background.

---

### Step 3: Add a Textual Indicator

Though our main focus is the circle, having a textual indicator can provide additional context. Let's add a simple “Breathe In” and “Breathe Out” text that will later be toggled through JavaScript as the circle expands and contracts.

Modify the HTML:

```
“html
<div class=“container”>
 <div class=“circle”>
 <p class=“text”>Breathe In</p>
```

```
</div>
</div>
```
Now, add some styling for the text:
```
css
.text {
 position: absolute;
 top: 50%;
 left: 50%;
 transform: translate(-50%, -50%);
 font-family: 'Arial', sans-serif;
 color: fff;
 font-size: 1.5rem;
 text-align: center;
 opacity: 0.8;
}
```

```

Conclusion

We have now successfully created a large circle which will serve as the core visual element for our Relaxer App. This circle will soon be animated to assist users in a relaxing breathing exercise. In the next chapters, we will dive deeper into adding animations and enhancing the interactivity of this circle.

Remember, the effectiveness of this app isn't just about the animations or the code, but also about the visual cues that can help users find relaxation and calmness. Our choice of colors, positioning, and the smoothness of animations will play a crucial role in the app's overall impact.

Creating & Animating The Pointer

In this chapter, we will be focusing on an integral part of the Relaxer App: the pointer. The pointer serves as a visual cue to guide the user through their breathing exercises. It will rotate around the circumference of a circle, with its motion corresponding to the breathing instructions.

Prerequisites

Before we begin, ensure you have the following set up:

- Basic HTML structure for the Relaxer App
 - A large circle in the center of the page (from Chapter 70)
-

1. Creating the Pointer with HTML & CSS

HTML:

Add a div for the pointer within the circle element:

```
“html
<div class=“circle”>
    <div class=“pointer”></div>
</div>
“
```

CSS:

Style the pointer to appear as a small, prominent triangle that stands out against the circle:

```
“css
.pointer {
    width: 0;
```

```
height: 0;  
border-left: 10px solid transparent;  
border-right: 10px solid transparent;  
border-bottom: 20px solid 333;  
position: absolute;  
top: -10px;  
left: 50%;  
transform: translateX(-50%);  
}  
“
```

2. Basic Animation Setup

To animate the pointer, we will be using CSS keyframes and the `transform` property. For simplicity, let's set up an animation that rotates the pointer 360 degrees over a duration of 8 seconds (to mimic a full breath cycle).

CSS:

```
“css  
@keyframes rotatePointer {  
    0% {  
        transform: rotate(0deg);  
    }  
    100% {  
        transform: rotate(360deg);  
    }  
}
```

Apply the animation to the pointer:

```
“css
```

```
.pointer {  
    /* ...previous styles... */  
    animation: rotatePointer 8s linear infinite;  
}  
“
```

3. Timing with `setTimeout`

While the CSS animation gives us a continuous motion, we want the pointer to pause during specific moments to simulate the ‘inhale’ and ‘exhale’ periods. For this, we’ll employ JavaScript’s `setTimeout` method.

JavaScript:

First, grab the pointer element:

```
“javascript  
const pointer = document.querySelector('.pointer');  
”
```

Next, create a function to control the pointer’s animation:

```
“javascript  
function controlPointerAnimation() {  
    // Pause the animation  
    pointer.style.animationPlayState = 'paused';  
    // 'Inhale' for 4 seconds  
    setTimeout(() => {  
        pointer.style.animationPlayState = 'running';  
    }, 4000);  
    // 'Exhale' for 4 seconds and then pause again  
    setTimeout(() => {  
        pointer.style.animationPlayState = 'paused';  
    }, 8000);
```

```
}
```

```
"
```

Invoke the function when the page loads:

```
"`javascript
controlPointerAnimation();
"``
```

For continuous cycles, you can use `setInterval` to run the `controlPointerAnimation` function every 8 seconds.

```
"`javascript
setInterval(controlPointerAnimation, 8000);
"``
```

Conclusion

The animated pointer now provides a clear visual cue for users, guiding them through the breathing exercise. By combining CSS animations with timed JavaScript functions, we've created a seamless user experience that's both interactive and beneficial for relaxation.

In the next chapter, we'll delve into triggering the breathing animation with JavaScript, enhancing the relaxation experience further. Stay tuned!

Breath Animation With JS Trigger

Objective: In this chapter, we'll walk through how to create a breathing relaxation app. The app will encourage users to follow a rhythmic breathing pattern, using a visual guide that expands and contracts. The animation will be achieved using CSS animations and triggered via JavaScript.

1. Introduction

Breathing exercises are often used to induce relaxation and reduce stress. Our goal is to create an animation that mimics the inhalation and exhalation process, guiding users to sync their breath with it.

2. Setting Up the HTML Structure

Let's start by creating a basic structure for our animation:

```
``html
<div class="app">
  <div class="circle">
    <div class="inner-circle"></div>
  </div>
  <p id="instruction">Breathe In</p>
</div>
``
```

Here, `circle` represents the breath visual while `inner-circle` mimics the lungs expanding and contracting.

3. Styling Our Animation

Next, let's style our circles:

```
``css
.app {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  background-color: e0f7fa;
}

.circle {
```

```
width: 100px;  
height: 100px;  
border-radius: 50%;  
background-color: 81d4fa;  
display: flex;  
justify-content: center;  
align-items: center;  
overflow: hidden;  
}  
.inner-circle {  
width: 50px;  
height: 50px;  
border-radius: 50%;  
background-color: 29b6f6;  
opacity: 0.7;  
}  
“
```

4. Creating the CSS Animation

Let's create two animations: one for breathing in, where the circle expands, and one for breathing out, where it contracts.

```
“css  
@keyframes breathIn {  
0%, 100% {  
width: 50px;  
height: 50px;  
}  
50% {
```

```
    width: 100px;  
    height: 100px;  
}  
}  
  
@keyframes breatheOut {  
    0%, 100% {  
        width: 100px;  
        height: 100px;  
    }  
    50% {  
        width: 50px;  
        height: 50px;  
    }  
}  
“
```

5. Adding JavaScript Trigger

Now, let's use JavaScript to trigger these animations:

```
``javascript  
const instruction =  
document.getElementById('instruction');  
  
const innerCircle = document.querySelector('.inner-  
circle');  
  
let isBreathingIn = true;  
  
const breatheFunction = () => {  
    if (isBreathingIn) {  
        instruction.innerText = 'Breathe Out';  
        innerCircle.style.animation = 'breatheln 2s  
alternate 2';
```

```
    } else {
        instruction.innerText = 'Breathe In';
        innerCircle.style.animation = 'breatheOut 2s
alternate 2';
    }
    isBreathingIn = !isBreathingIn;
};

// Initial call
breatheFunction();
// Use setInterval to keep repeating
setInterval(breatheFunction, 4000);
``
```

Here, we're toggling between the `breathIn` and `breatheOut` animations every 4 seconds, giving each animation 2 seconds to complete with a 2-second pause (achieved by using the `alternate` count).

6. Conclusion

With this, our breath animation is complete. The `Relaxer App` now provides a visual aid for users to synchronize their breathing. CSS animations combined with JavaScript triggers offer a dynamic way to provide engaging content for users.

By following this model, you can expand upon this concept to include sound, adjust timings, or even include varying breathing patterns for different relaxation techniques.

Note: Always ensure your applications, especially those with health implications, have appropriate disclaimers. This app is a basic model and should be used for illustrative purposes. It's always best to consult with

professionals when dealing with health and well-being related content

Section 14: Project 13 - New Year Countdown | DOM, Date & Time

Project Intro

Welcome to the thirteenth project of this book! We have come a long way, from form validators to custom video players, and now we're diving into a delightful New Year Countdown application. As we bid goodbye to the current year and eagerly await the next, having a visual countdown can heighten the excitement and thrill. Whether you want to deploy this on your website, integrate it into an event page, or simply enjoy the programming journey, this project promises to offer both challenges and fulfillment.

Objective of the Project

The primary objective of this project is to create a web-based New Year countdown timer. As seconds tick away and the new year approaches, our application will provide real-time updates, showing the days, hours, minutes, and seconds remaining.

Key Features

1. Dynamic Date & Time Integration: Using the JavaScript Date object, the application will continually fetch the current date and time, calculate the difference from the New Year, and display the countdown.

2. Landing Page: A beautifully designed landing page that captivates users with animations, relevant images, and, of course, the countdown itself.
3. Responsive Design: We want everyone, regardless of their device, to enjoy the countdown. Hence, we'll make sure our design is responsive, looking great on both mobiles and desktops.
4. Spinner Effect: Just before the countdown hits the New Year, we'll have a spinner effect to add to the anticipation.
5. Dynamic Year Update: No need to manually update the target year. Our app will automatically set the countdown for the next New Year.

Technologies & Methods to be Used

- * HTML: For structuring the landing page and countdown elements.
- * CSS: To style, beautify, and ensure responsiveness of our application. We will also use CSS for creating animations.
- * JavaScript: This is where the magic happens. We'll use JS to:
 - Fetch the current date and time.
 - Calculate the difference between the current date and the New Year.
 - Update the DOM elements in real-time.
 - Implement the spinner effect and dynamic year updates.

What You Will Learn

By the end of this project:

- You'll gain a deeper understanding of the JavaScript Date object and its methods.

- You'll learn to manipulate the DOM in real-time, making your applications more dynamic and interactive.
- The importance of responsiveness in design will be clearer, and you'll have practical knowledge of implementing it.
- You'll experience the thrill of building an application that's time-sensitive.

Prerequisites

Before diving into this project, ensure you have a basic understanding of HTML, CSS, and JavaScript. Previous projects in this book have covered DOM manipulation, CSS styling, and JavaScript fundamentals, which will be critical to succeeding in this project.

In the next chapter, we'll begin our journey by setting up our landing page. This will be the foundation upon which we'll build our dynamic countdown. So, make sure you have your code editor ready, your favorite browser open, and let's welcome the New Year with some code!

Landing Page HTML & Styling

In this chapter, we'll focus on creating the landing page for our New Year Countdown project. The goal is to design a visually appealing webpage that also provides the functional backdrop for our countdown timer. We'll be using HTML for the structure and CSS for styling.

1. Setting up the HTML Structure

For the landing page, we want a simple yet vibrant design that builds anticipation for the New Year countdown.

Let's begin with the basic HTML structure:

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>New Year Countdown</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <header>
            <h1>New Year Countdown</h1>
            <p>Welcome the new year with excitement!</p>
        </header>
        <div class="countdown-wrapper">
            <!-- Countdown Timer will be placed here in the
next chapter -->
        </div>
    </div>
</body>
</html>
``
```

Here, we've set up a `container` div to hold the entire content. Inside this, the `header` consists of a heading and a subheading. The `countdown-wrapper` is a placeholder for where we'll be adding our countdown timer in the next chapter.

2. Styling the Landing Page

Let's now move on to styling. For this project, let's consider a dark theme with gold accents to give it a festive feel.

styles.css

```
``css
body {
    font-family: 'Arial', sans-serif;
    background-color: #222;
    color: #fff;
    text-align: center;
    margin: 0;
    padding: 0;
    height: 100vh;
    display: flex;
    justify-content: center;
    align-items: center;
}
.container {
    width: 80%;
    max-width: 800px;
}
header {
    margin-bottom: 20px;
}
header h1 {
    font-size: 2.5rem;
    color: gold;
    margin: 0;
```

```
}

header p {
    color: ddd;
    font-style: italic;
}

.countdown-wrapper {
    border: 2px solid gold;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0px 0px 15px rgba(255, 215, 0, 0.6);
}

``
```

Here's a breakdown of what we did with our CSS:

- For the body, we used a dark color (`222`) and set the text color to white. We also centered our `container` using flexbox properties.
- The container width is set to 80% of the viewport with a max-width of 800px to ensure our design is responsive.
- The header has a bottom margin to give space between it and the countdown timer. The h1 tag (title) is colored gold to stand out.
- We've styled the `countdown-wrapper` with a golden border and added a subtle box shadow to give it some depth.

3. Conclusion

With our landing page's HTML and styling complete, we now have a base from which we can implement our countdown timer in the upcoming chapters. The dark theme accented with gold adds a touch of festivity and anticipation for the New Year's countdown.

In the next chapter, we'll focus on creating the actual countdown, adding days, hours, minutes, and seconds dynamically, and using JavaScript to make it all function in real-time.

Create The Countdown

Chapter 75: Create The Countdown

Welcome to Chapter 75 of our project journey! In this chapter, we're diving into the heart of our New Year Countdown project: creating the countdown itself. We will fetch the current date, determine the New Year's date, and calculate the difference between the two. The result will then be displayed dynamically on our landing page.

1. Understanding the Objective

Our goal is to display a countdown that decrements every second and shows the number of days, hours, minutes, and seconds left until the New Year. This dynamic visual engages users and builds anticipation for the upcoming year.

2. Setting up the HTML Structure

Before diving into the JavaScript, let's create a placeholder in our HTML where our countdown will be displayed:

```
``html
<div class="countdown">
    <div class="time">
        <span class="days">365</span> days
    </div>
    <div class="time">
        <span class="hours">00</span> hours
    </div>
</div>
```

```
</div>
<div class="time">
    <span class="minutes">00</span> minutes
</div>
<div class="time">
    <span class="seconds">00</span> seconds
</div>
</div>
``
```

3. Getting Current Date & New Year Date

In JavaScript, we need to get the current date and the date of the next New Year:

```
```javascript
const currentDate = new Date();
const nextNewYearDate = new
Date(currentDate.getFullYear() + 1, 0, 1);
``
```

---

### 4. Calculate the Difference

We need to calculate the difference in milliseconds between the current date and the New Year's date:

```
```javascript
const difference = nextNewYearDate - currentDate;
``
```

From this difference, we can derive the days, hours, minutes, and seconds.

5. Deriving Time Units

Here's a breakdown of how to derive each time unit:

```
``javascript
const seconds = Math.floor(difference / 1000);
const minutes = Math.floor(seconds / 60);
const hours = Math.floor(minutes / 60);
const days = Math.floor(hours / 24);
const secondsLeft = seconds % 60;
const minutesLeft = minutes % 60;
const hoursLeft = hours % 24;
``
```

6. Displaying the Countdown

Now, we will update our HTML structure with the calculated values:

```
``javascript
document.querySelector('.days').textContent = days;
document.querySelector('.hours').textContent =
hoursLeft;
document.querySelector('.minutes').textContent =
minutesLeft;
document.querySelector('.seconds').textContent =
secondsLeft;
``
```

7. Updating the Countdown Every Second

To make our countdown dynamic, we need to update it every second:

```
``javascript
setInterval(() => {
  // Recalculate the difference
  const currentDate = new Date();
``
```

```
const difference = nextNewYearDate - currentDate;  
// Convert the difference to days, hours, minutes, and  
seconds  
// ...  
// Update the HTML  
// ...  
, 1000);  
``
```

And that's it! You've successfully created a dynamic New Year countdown using DOM manipulation and the Date object in JavaScript.

Closing Thoughts

Countdowns can be a thrilling addition to many websites, especially those anticipating an event. By mastering the Date object and understanding the mathematics behind time calculations, you can create a plethora of time-based functionalities for the web. As always, practice makes perfect. Try creating countdowns for other events or adding features like custom user inputs for personalized countdowns. The sky's the limit!

Dynamic Year & Spinner

Welcome back to Project 13, the New Year Countdown! In our previous chapter, we laid the groundwork by setting up our landing page. This time, we'll delve deeper into the JavaScript world and implement the dynamic year display and a spinner animation, showcasing the countdown to the New Year.

Setting Up the Dynamic Year

One of the most essential features in a New Year Countdown is ensuring that it remains relevant year after

year. Instead of hardcoding the year, we can utilize JavaScript's `Date` object to make sure our countdown is always counting down to the upcoming New Year.

Step 1: Access the DOM Elements

Before we start, we need references to the elements where we want to display the year.

```
``javascript
const yearEl = document.getElementById('year');
``
```

Step 2: Get the Current Year

Using the `Date` object, we can fetch the current year:

```
``javascript
const currentYear = new Date().getFullYear();
``
```

Step 3: Display the Next Year

```
``javascript
yearEl.textContent = currentYear + 1;
``
```

This will ensure that the displayed year is always the next one, keeping our countdown relevant.

Implementing the Spinner Animation

As users visit our page, it's a pleasant experience to provide them with a visual cue indicating that the countdown is being set up. This is where our spinner comes into play. It will display for a short time before revealing the countdown.

Step 1: HTML Setup

Within your main HTML, incorporate the spinner div:

```
``html
<div class="spinner" id="spinner"></div>
```

Step 2: CSS Styling

For the spinner, you'll want a simple rotating animation:

```
```
.css
.spinner {
 width: 50px;
 height: 50px;
 border: 5px solid rgba(255, 255, 255, 0.1);
 border-top: 5px solid fff;
 border-radius: 50%;
 position: absolute;
 top: 50%;
 left: 50%;
 transform: translate(-50%, -50%);
 animation: spin 1s linear infinite;
}

@keyframes spin {
 0% {
 transform: translate(-50%, -50%) rotate(0deg);
 }
 100% {
 transform: translate(-50%, -50%) rotate(360deg);
 }
}
```

```

Step 3: JavaScript Logic

Finally, to ensure that the spinner only shows for a brief time before the countdown is displayed:

```
``javascript
const spinnerEl = document.getElementById('spinner');
const countdownEl =
document.getElementById('countdown'); // Assuming
this is the id of your countdown container
setTimeout(() => {
  spinnerEl.style.display = 'none';
  countdownEl.style.display = 'block';
}, 1000);
``
```

This will hide the spinner and display the countdown after a second.

Wrapping Up

With our dynamic year in place and a pleasant loading spinner to greet our users, our New Year Countdown is taking shape!

Remember, the power of web development is not just in creating beautiful visuals but in enhancing user experience with these small dynamic functionalities. The countdown becomes more robust and user-friendly, ensuring our users return year after year.

Section 15: Project 14 - Sortable List | Drag & Drop API

Project Intro

Welcome to Project 14! In this module, we are going to explore one of the cool features of modern web

development — the Drag & Drop API. As web developers, we're always on the lookout for ways to enhance user experience and make web interactions feel natural and intuitive. The Drag & Drop feature is a versatile tool that, when implemented well, can significantly elevate the functionality and usability of a website or application.

Objective of the Project

Our primary goal in this project is to develop a sortable list. Think of a scenario where you have a list of tasks, priorities, or perhaps even a playlist of songs, and you want to rearrange them based on their importance, urgency, or preference. Wouldn't it be amazing if you could simply drag an item and place it wherever you wanted within the list? That's exactly what we're going to build in this project.

Why a Sortable List?

The ability to manipulate the order of items in a list through dragging and dropping is not just visually satisfying for the user, but it's also extremely functional. Here's why:

1. **Flexibility:** Users are not always sure of the order of items when they initially create a list. Having the ability to reorder items provides the flexibility to adapt the list over time.
 2. **Intuitiveness:** Dragging and dropping feels natural. It mimics the action of physically moving objects around, making the interaction feel real and immersive.
 3. **Enhanced User Experience:** No need to manually delete and re-add items to change their order. A sortable list provides a seamless way to organize items.
-

Technical Overview

In this project, we'll harness the capabilities of the HTML5 Drag & Drop API. Here are the primary components we will be focusing on:

1. Draggable Elements: These are the items in our list that we want to make movable.
 2. Drop Zones: These are the areas where we can place our draggable elements.
 3. Drag & Drop Events: These are a set of events that the browser triggers as elements are dragged and dropped. We'll be leveraging events like `dragstart`, `dragover`, `dragleave`, and `drop`.
-

Expected Outcomes

By the end of this project:

- You'll have a functional sortable list that allows items to be rearranged using drag and drop.
 - You'll gain a deep understanding of the Drag & Drop API, its events, and methods.
 - You'll be able to implement drag & drop functionality in different scenarios, not just for sorting lists.
-

Prerequisites

While we will be explaining each step in detail, it's essential to have a basic understanding of:

- HTML: To structure our list.
 - CSS: For styling our list and creating visual cues during the drag & drop process.
 - JavaScript: To handle drag & drop events and manipulate the DOM.
-

In Conclusion

The Drag & Drop API opens up a world of interactive possibilities on the web. Whether it's building a game, a

planner, or a visual organizer, understanding how to harness its power can set your projects apart.

Are you excited? Let's dive right into building our sortable list!

Insert List Items Into DOM

In this chapter, we will explore one of the foundational tasks in web development – inserting list items into the Document Object Model (DOM). The ability to dynamically insert elements into the DOM is vital for creating dynamic web applications. In the context of our sortable list project, we will be creating a list that the user can interact with by dragging and dropping list items. But before we can do that, we need to get our list items into the DOM. Let's get started!

Understanding the DOM

Before diving into the insertion process, it's essential to understand what the DOM is. The DOM represents the structure of your web document, and every element, attribute, and piece of text becomes a node within this structure. When we talk about inserting list items into the DOM, we are essentially adding nodes to this structure.

Basic List Structure

In HTML, a basic list structure using unordered lists looks like this:

```
``html
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

“

In our sortable list project, we want to dynamically insert these list items.

Inserting List Items Using JavaScript

Here is a step-by-step guide to inserting list items into the DOM using JavaScript:

1. Select the Parent Element

Before we can insert a list item, we need to select the parent element. In our case, it's the `` element.

```
``javascript
const list = document.querySelector('ul');
````
```

### 2. Create the List Item Element

We can create a new element using the `document.createElement()` method.

```
``javascript
const listItem = document.createElement('li');
````
```

3. Set the Content for the List Item

You can set the text content for this newly created list item using the `textContent` property.

```
``javascript
listItem.textContent = 'Item 4';
````
```

### 4. Append the List Item to the Parent Element

To insert the new list item into the DOM, you use the `appendChild()` method.

```
``javascript
list.appendChild(listItem);
````
```

```
``
```

If you followed the above steps, your list will now look like this:

```
``html
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
</ul>
``
```

Dynamically Inserting Multiple List Items

To make our project interactive, we might want to insert multiple list items at once. For example, if we have an array of items, we can iterate over this array and insert each item into the DOM:

```
``javascript
const items = ['Item 5', 'Item 6', 'Item 7'];
items.forEach(item => {
  const listItem = document.createElement('li');
  listItem.textContent = item;
  list.appendChild(listItem);
});
```

This will extend our list to:

```
``html
<ul>
  <li>Item 1</li>
```

```
<li>Item 2</li>
<li>Item 3</li>
<li>Item 4</li>
<li>Item 5</li>
<li>Item 6</li>
<li>Item 7</li>
</ul>
“
```

Conclusion

Being able to manipulate the DOM is a fundamental skill for a web developer. By now, you should have a good understanding of how to insert list items into the DOM dynamically using JavaScript. In the next chapter, we'll learn how to scramble these list items and set the stage for our drag-and-drop functionality. Stay tuned!

Scramble List Items

In this chapter, we'll introduce the concept of scrambling the items in our sortable list. The aim is to challenge the user to rearrange the list into its original order. By doing this, we'll provide a real-world application of the Drag & Drop API, as it demonstrates a practical use-case.

Understanding the Need for Scrambling

Before we dive into the technicalities, let's understand the need for scrambling. When we provide a list to users in a sorted order, and then task them with rearranging it after it's been scrambled, we're giving them a simple, yet effective exercise in sorting. This could be applied in real-world scenarios like sorting tasks, rearranging products in a list, or any other kind of ordering requirement.

Getting Started

First, we'll assume that we have a list of items, structured in an HTML format as follows:

```
``html
<ul id="sortable-list">
    <li draggable="true">Item 1</li>
    <li draggable="true">Item 2</li>
    <li draggable="true">Item 3</li>
    <!--... and so on -->
</ul>
``
```

JavaScript: Scrambling the List

We'll use a modern version of the Fisher-Yates (also known as the Knuth) shuffle algorithm. This algorithm ensures each permutation of the list is equally likely.

```
``javascript
function scrambleList() {
    const ul = document.querySelector('sortable-list');
    const items = ul.getElementsByTagName('li');
    let itemsArr = Array.prototype.slice.call(items);
    itemsArr.sort(() => 0.5 - Math.random());
    itemsArr.forEach(item => {
        ul.appendChild(item);
    });
}
```

In this function:

- We first select our unordered list by its ID.

- We get all the list items from the unordered list.
 - We convert the HTML collection to an array, so we can use array methods on it.
 - We then use the `sort()` method to shuffle the list. The logic `0.5 - Math.random()` is a commonly used trick to randomize the sort.
 - Finally, we append each shuffled item back to the list.
-

Executing the Scramble

Now, you'll probably want to trigger the scramble at a specific point in your application. For demonstration purposes, we'll just add a button which, when clicked, will scramble our list.

```
``html
<button onclick="scrambleList()">Scramble
Items</button>
``
```

Conclusion

By introducing a scrambling feature, you've added an interactive layer to your sortable list application. This challenges the users, but also gives them a tangible goal: to restore order. In the next chapter, we'll delve into the core styling of our application and set the visual stage for our drag and drop functionality. Remember, user experience is paramount, and the visual feedback we provide is crucial in guiding users through the sorting process.

Core CSS

CSS, or Cascading Style Sheets, is an essential tool for controlling the look and feel of your web applications. In this chapter, we'll be diving into the core CSS that will give our Sortable List project a clean, professional

appearance. Remember, good CSS not only makes your application visually appealing, but it can also enhance user experience.

1. Resetting Default Styles:

Before diving into the project-specific styles, it's essential to ensure that we reset browser-default styles. This will give us a consistent baseline across different browsers.

```
``css
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
``
```

2. Setting up the Body & Main Wrapper:

The `body` and main wrapper elements dictate the basic styling and positioning of our entire app.

```
``css
body {
    font-family: 'Arial', sans-serif;
    background-color: f4f4f4;
    color: 333;
    font-size: 16px;
    line-height: 1.6;
}
.wrapper {
    width: 80%;
    max-width: 1200px;
```

```
margin: 40px auto;  
padding: 20px;  
background-color: fff;  
box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);  
}  
``
```

3. Styling the Sortable List:

For our sortable list, we'll aim for a design that differentiates each list item while making it clear that they can be dragged and dropped.

```
``css  
.sortable-list {  
    list-style-type: none;  
}  
.sortable-list li {  
    padding: 15px;  
    border: 1px solid ddd;  
    cursor: move; /* Indicates draggability */  
    margin-bottom: 10px;  
    background-color: f9f9f9;  
    transition: background-color 0.2s;  
}  
.sortable-list li:hover {  
    background-color: f4f4f4;  
}  
``
```

4. Drag & Drop Indication:

When a user drags a list item, we'll want to give a visual indication. A simple way is to change the background color.

```
``css
.sortable-list li.dragging {
    background-color: ddd;
    border: 1px solid transparent;
}
``
```

5. Feedback and Notifications:

If you've integrated any feedback mechanism like an alert to indicate the correct order, we'll want to style it accordingly.

```
``css
.alert {
    padding: 10px;
    margin: 10px 0;
    border: 1px solid transparent;
    border-radius: 5px;
}
.alert.success {
    color: 006400;
    background-color: e6ffe6;
    border-color: 006400;
}
.alert.error {
    color: a80000;
    background-color: ffe6e6;
```

```
border-color: a80000;  
}  
“
```

6. Additional Touches:

Finally, you can add some additional touches to improve the aesthetics and user experience. Consider adding `hover` effects, transitions, and possibly some responsive media queries to ensure your sortable list looks good on all devices.

Conclusion:

CSS is a powerful tool that enables web developers to create visually appealing and intuitive user interfaces. By using thoughtful and purposeful styles, you can significantly improve user experience. In the next chapters, we'll integrate the drag and drop API functionality into our styled sortable list, adding functionality to our visually appealing design.

Remember, always test your designs on various devices and browsers to ensure consistency and optimal user experience.

Drag & Drop Functionality

Drag and Drop (DnD) functionality is a user interface pattern that allows users to directly manipulate elements on the screen, offering an intuitive way to rearrange elements, move data between containers, or even create elements by dragging a new object into a designated space. In the context of our Sortable List project, we'll use this feature to rearrange list items by dragging them to their desired locations.

Basics of the HTML Drag and Drop API

Before diving into the code, it's crucial to understand the foundational concepts of the Drag and Drop API:

- **draggable attribute:** This attribute is set on the element you wish to drag. Most HTML elements are not draggable by default, so you need to set `draggable="true"` on them.

```
``html
<div draggable="true">This is a draggable element.
</div>
``
```

- **Drag & Drop Events:** These are the primary events that the DnD API provides, which give us control over the drag and drop lifecycle.

- **dragstart:** Triggered when the user starts dragging an element.

- **drag:** Triggered as the element is being dragged.

- **dragend:** Triggered when the user releases the element.

- **dragenter:** Triggered when a dragged element enters a drop target.

- **dragover:** Triggered as the dragged element is over a drop target (many times).

- **dragleave:** Triggered when a dragged element leaves a drop target.

- **drop:** Triggered when the dragged element is dropped on a drop target.

Implementing Drag & Drop for our Sortable List

1. Setting Items as Draggable

Every list item that you want to be draggable should have the `draggable` attribute set to `true`. Assuming you have a list like this:

```
``html
```

```
<ul id="sortable-list">  
  <li draggable="true">Item 1</li>  
  <li draggable="true">Item 2</li>  
  <!-- ... -->  
</ul>  
``
```

2. Add Event Listeners

Now, we'll attach event listeners to handle the drag and drop operations.

```
```javascript  
const listItems = document.querySelectorAll('sortable-list li');

listItems.forEach(item => {
 item.addEventListener('dragstart', handleDragStart);
 item.addEventListener('dragover',
 handleDragOver);
 item.addEventListener('drop', handleDrop);
 item.addEventListener('dragleave',
 handleDragLeave);
});
``
```

## 3. Handling Drag Start

When dragging starts, we'll set some data on the event to recognize what's being dragged.

```
```javascript  
function handleDragStart(e) {  
  e.dataTransfer.setData('text/plain', e.target.id);  
  setTimeout(() => {  
    e.target.classList.add('hidden');  
  }, 0);  
}
```

```
}
```

```
"
```

The `hidden` class can be something simple that visually hides the item, like:

```
``css
```

```
.hidden {  
    display: none;  
}
```

```
"
```

4. Handling Drag Over

By default, dropping is disabled on most elements. To allow a drop, we need to prevent the default handling of the event.

```
``javascript
```

```
function handleDragOver(e) {  
    e.preventDefault();  
    e.target.classList.add('over');  
}
```

```
"
```

The `over` class can be a styling indication for the placeholder where the element will be dropped.

5. Handling Drop

This is where the magic happens! Here, we'll rearrange our list based on where the item was dropped.

```
``javascript
```

```
function handleDrop(e) {  
    e.preventDefault();  
    const draggedId =  
        e.dataTransfer.getData('text/plain');
```

```
const draggedItem =  
document.getElementById(draggedId);  
  
const dropTarget = e.target;  
  
// Insert the dragged item before the drop target  
dropTarget.parentNode.insertBefore(draggedItem,  
dropTarget);  
  
draggedItem.classList.remove('hidden');  
  
}  
  
``
```

6. Handling Drag Leave

When the draggable item leaves a potential drop target, we want to remove any indication that the drop target is active.

```
``javascript  
function handleDragLeave(e) {  
    e.target.classList.remove('over');  
}  
``
```

Conclusion

The Drag and Drop API provides a powerful way to implement intuitive interactions in your web projects. With this API, we can create visually appealing interfaces where users can directly manipulate on-screen elements, making for a more dynamic and engaging user experience.

Check Order

After mastering the art of dragging and dropping items on our sortable list, the next vital step is to check the order of the items. This ensures that users receive

feedback on their effort, thereby making the sortable list interactive and intuitive.

In this chapter, we'll focus on determining the correct order of the items after they've been sorted. We'll walk through creating a function to validate the order, and then provide feedback to the user based on their sorting performance.

Prerequisites

Before delving into the code, ensure that:

- You have a working drag and drop functionality from the previous chapters.
 - You have a predetermined correct order for the list items to compare against.
-

Setting Up the Correct Order

For the sake of our tutorial, let's imagine our list represents the order of planets from the sun:

```
``javascript
const correctOrder = ['Mercury', 'Venus', 'Earth', 'Mars',
  'Jupiter', 'Saturn', 'Uranus', 'Neptune'];
``
```

Checking the Order

Once a user finishes sorting the list, we need to check the order against our `correctOrder` array.

```
``javascript
function checkOrder() {
  const listItems = document.querySelectorAll('li');
  let isCorrect = true;
  listItems.forEach((item, index) => {
    if (item.textContent !== correctOrder[index]) {
```

```
    isCorrect = false;  
  }  
});  
return isCorrect;  
}  
``
```

In this function:

- We first select all the `li` elements.
- We initialize a variable `isCorrect` to `true`.
- We then loop through each list item and compare its text content with the correct order. If any do not match, we set `isCorrect` to `false`.

Providing Feedback

Once we've determined whether the order is correct or not, it's essential to provide feedback to the user.

```
``javascript  
function provideFeedback() {  
  const isOrderCorrect = checkOrder();  
  if (isOrderCorrect) {  
    alert("Congratulations! You've sorted the list  
correctly.");  
  } else {  
    alert("Oops! That doesn't seem right. Try again.");  
  }  
}
```

Now, we'll call `provideFeedback` every time a list item is dropped.

Hooking Up the Feedback Function to Drag & Drop

Update your `drop` event listener:

```
``javascript
listItem.addEventListener('drop', function(e) {
    // ... existing drag & drop logic ...
    provideFeedback();
});``
```

Now, every time an item is dropped, the order is checked, and feedback is provided.

Enhancements

1. **Highlight Incorrect Items:** Instead of a simple alert, you can add a red border or change the background color of incorrect items to visually guide users.
2. **Add a “Check Order” Button:** Instead of checking immediately upon every drop, you could add a button that allows users to check their order when they feel they’ve sorted it correctly.

Conclusion

Incorporating feedback mechanisms, such as checking the order in our sortable list, greatly enhances the user experience. It transforms the list from a simple sorting activity into a more interactive and engaging puzzle. Always remember, providing real-time feedback is crucial in any interactive web application, ensuring users remain engaged and informed about their actions.

As you progress with web development, consider diving deeper into the Drag & Drop API to explore other possibilities, like multi-list dragging, clone dragging, and more!

Section 16: Project 15 - Breakout Game | HTML5 Canvas API

Project Intro

Welcome to Project 15, the “Breakout Game” using the HTML5 Canvas API! If you’ve ever had a taste of classic arcade games, you’re in for a delightful trip down memory lane. If not, prepare to experience one of the gems of yesteryears that laid the foundation for many modern games.

The Breakout Game

The Breakout Game, often simply termed “Breakout”, has a simple premise. You control a paddle, moving it left or right, with the aim to bounce a ball that ricochets off walls and breaks a collection of bricks placed overhead. As you break the bricks, they disappear, and the objective is to clear all the bricks without letting the ball fall past your paddle. Sounds simple, right? Well, with each level, the game can introduce faster ball speeds, different brick layouts, and other challenges to keep you on your toes!

Why This Project?

This game offers the perfect platform to dive deep into the HTML5 Canvas API. Through this project, you’ll understand:

- How to draw and animate shapes on the canvas.
- Implementing logic to detect collisions between the ball and the bricks or paddle.

- Capturing user inputs to move the paddle.
 - Manipulating the canvas in real-time to reflect game states like scores, ball resets, or game over scenarios.
-

What Will You Learn?

By the end of this project, you'll be familiar with:

1. HTML5 Canvas Basics: Setting up a canvas, understanding its coordinate system, and drawing basic shapes.
 2. Animation: Using `requestAnimationFrame` to create smooth animations.
 3. Collision Detection: Understanding how to detect when the ball hits the paddle, the walls, or the bricks.
 4. User Input Handling: Capturing and processing keyboard inputs to move the paddle.
 5. Game Logic: Implementing the game's core mechanics, like ball movement, score tracking, and level progression.
-

Prerequisites

Before we start, make sure you're familiar with basic JavaScript concepts, as we'll be writing game logic in JS. Knowledge of arrays, loops, and conditional statements will be beneficial. Don't worry if you're new to the Canvas API; we'll cover everything step-by-step.

Tools & Technologies

- HTML: To set up our game's structure and canvas.
- CSS: A bit of styling to make our game look appealing.
- JavaScript: The core of our game logic. We'll be using vanilla JS, so no external libraries are required.
- HTML5 Canvas API: To draw and animate our game elements.

Wrapping Up

This project promises to be both fun and educational. By its end, not only will you have a functional game that you can show off to friends and family, but you'll also have acquired a deeper understanding of game development principles, the Canvas API, and JavaScript. So, roll up your sleeves, and let's dive into the world of game development with our Breakout game!

In the next chapter, we'll start by setting up our game's page and styling. Get ready, and let's break some bricks!

Creating & Styling The Page

In this chapter, we will embark on the initial step of our Breakout Game project. Before diving into the HTML5 Canvas API and creating the actual game mechanics, we first need to lay the groundwork. A well-structured and styled webpage will be the foundation for our game. Let's get started!

1. Setting Up the HTML Structure:

To begin with, our HTML structure will be simple. We need a title for our game, a canvas element where the game will be rendered, and possibly a footer or section for game instructions or credits.

```
``html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Breakout Game</title>
```

```
<link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Breakout Game</h1>
    </header>
    <section id="game-area">
        <canvas id="breakoutCanvas" width="800"
height="600"></canvas>
    </section>
    <footer>
        <p>Instructions: Use your arrow keys or touch
controls to move the paddle. Break all bricks to win!</p>
    </footer>
</body>
</html>
```

“

Note:

- The `canvas` element has been given a fixed width and height. These dimensions can be adjusted based on your design preferences.
- We've linked an external CSS file named `styles.css` to style our game page.

2. Styling the Page:

For our game's visual appeal, we'll opt for a clean, centered layout with a subdued background so that the game area remains the primary focus.

styles.css:

“css

```
body {  
    font-family: 'Arial', sans-serif;  
    background-color: 282c34;  
    color: ffffff;  
    text-align: center;  
    display: flex;  
    flex-direction: column;  
    align-items: center;  
    justify-content: center;  
    height: 100vh;  
    margin: 0;  
}  
  
header {  
    margin-bottom: 20px;  
}  
  
h1 {  
    font-size: 2.5em;  
    margin: 0;  
}  
  
game-area {  
    position: relative;  
}  
  
canvasbreakoutCanvas {  
    border: 2px solid ffffff;  
    background-color: 1a1d23;  
}  
  
footer {  
    margin-top: 20px;
```

```
}
```

```
footer p {
```

```
    font-size: 0.9em;
```

```
}
```

```
"
```

Highlights:

- We've given our page a dark background (`282c34`) to help the colorful game elements pop.
- The game's canvas has a slightly different shade of dark (`1a1d23`) and is bordered for distinction.
- Our entire layout is centered both horizontally and vertically using Flexbox, providing a modern and sleek appearance.

3. Improvements & Enhancements:

While our base structure and style are set up, always consider ways you can enhance the user experience:

- You might want to add a button to start the game or a scoreboard to track the player's progress.
- Responsive designs can ensure the game looks great on all devices. Use media queries to adjust styling for smaller screens.
- Consider adding game sounds or background music for a more immersive experience.

Summary:

With our page structure in place and styles applied, we've created a solid foundation for our Breakout game. This ensures our game mechanics, which we'll dive into in the next chapters, will be showcased on a polished and professional-looking stage.

Remember, the design is a critical part of game development. It's not just about how it looks but also how

it feels. So, always take the time to refine and perfect the environment in which your game operates. In the following chapters, we'll bring our Breakout game to life using the HTML5 Canvas API. Stay tuned!

Canvas Plan Outline

Welcome to Chapter 85, where we'll be diving deep into the planning phase of our Breakout Game project. The canvas is a pivotal part of our game, and understanding its structure and functionalities is crucial for the game's development.

1. Introduction to Canvas

Before we jump into the specifics of our Breakout game, let's briefly recap what the Canvas is. The HTML5 `<canvas>` element is used to draw graphics, on the fly, via scripting (usually JavaScript). It allows for dynamic, scriptable rendering of 2D shapes and bitmap images.

2. Setting Up Our Canvas

To initiate our canvas, we'll be defining it within our HTML:

```
``html
<canvas id="breakoutCanvas" width="800"
height="600"></canvas>
``
```

This will create a drawing surface 800 pixels wide and 600 pixels tall. But to bring life to this canvas, we must use JavaScript.

In your JavaScript, target the canvas element:

```
``javascript
const canvas =
document.getElementById('breakoutCanvas');
```

```
const ctx = canvas.getContext('2d');
```

```
"
```

`getContext('2d')` gives us a drawing context on the canvas, which we'll use for the Breakout game's visual elements.

3. Elements of Our Breakout Game

Our Breakout game comprises the following elements:

- Ball
- Paddle
- Bricks
- Score Display
- Life Display

Each element will be drawn and manipulated using the canvas API.

4. Coordinate System

The canvas has a built-in coordinate system. The top-left corner of the canvas is defined as point (0,0). All elements are placed relative to this point. As we go downwards, the y-coordinate increases; as we go to the right, the x-coordinate increases.

5. Ball Dynamics

- Initial Position: Center of the canvas.
- Movement: The ball will move both in the x and y directions.
- Collision: We will need to detect collisions with the paddle, the bricks, and the canvas edges.

6. Paddle Dynamics

- Initial Position: Centered horizontally at the bottom of the canvas.
 - Movement: The paddle will only move horizontally, controlled by the user.
 - Width and Height: These dimensions will be constants that we can tweak as per our game's requirement.
-

7. Brick Configuration

- Rows and Columns: We'll design a pattern of bricks using multiple rows and columns.
 - Brick Size: Each brick will have a uniform size, but we can adjust this to fit our design.
 - Gaps: There will be small gaps between bricks to distinguish them clearly.
-

8. Score & Life Display

- Position: Top-left corner of the canvas.
 - Dynamics: The score will increment when the ball hits a brick, and the life count will decrement when the ball misses the paddle.
-

9. Color Scheme

A consistent and visually pleasing color scheme is essential. For our game, we'll be choosing:

- Ball: Red
 - Paddle: Black
 - Bricks: A gradient of blues (this will add a dynamic feel)
 - Background: Light grey
 - Score & Life Display: Dark grey with white text
-

10. Future Enhancements (Optional)

Planning ahead, we can think of additional features:

- Multiple levels with increasing difficulty
 - Power-ups that appear randomly
 - Different brick types, some harder to break
-

Conclusion

The canvas offers a plethora of opportunities for web-based game development. By understanding its capabilities and planning our game's structure, we set the stage for smooth development in subsequent chapters. In the upcoming chapters, we'll delve into the specifics of each game element, starting with drawing the ball and paddle on our canvas. Stay tuned!

Remember, planning is a crucial phase in any development process. A well-thought-out plan can help streamline the coding phase and reduce potential errors. Always consider spending ample time in this phase before jumping into coding.

Draw Ball, Paddle & Score

In this chapter, we will start to see our Breakout game take shape. We'll be diving deep into the HTML5 Canvas API, understanding how to draw and animate a ball, paddle, and keep score. Let's get started!

1. Setting Up the Canvas Context

Before we start drawing, let's set up our canvas context. We've already created our canvas element in our HTML, so we'll grab that and set its 2D rendering context:

```
``javascript
const canvas =
document.getElementById('breakoutCanvas');
const ctx = canvas.getContext('2d');
```

```
“
```

2. Defining the Ball Properties

We'll start by defining properties for our ball:

```
```
const ball = {
 x: canvas.width / 2,
 y: canvas.height - 30,
 radius: 10,
 speed: 2,
 dx: 2,
 dy: -2,
 color: 'blue'
};
```

```
```
“
```

Here, `dx` and `dy` represent the ball's movement along the x and y axis.

3. Drawing the Ball

Now, let's create a function to draw our ball:

```
```
function drawBall() {
 ctx.beginPath();
 ctx.arc(ball.x, ball.y, ball.radius, 0, Math.PI * 2);
 ctx.fillStyle = ball.color;
 ctx.fill();
 ctx.closePath();
}
```

```
```
“
```

This function uses the `arc()` method which draws a circle using the properties we set for our ball.

4. Defining the Paddle Properties

Let's define our paddle:

```
``javascript
const paddle = {
    height: 10,
    width: 75,
    x: (canvas.width - 75) / 2,
    color: 'green'
};``
```

We've set the initial position of the paddle to be centered at the bottom of the canvas.

5. Drawing the Paddle

Here's our function to draw the paddle:

```
``javascript
function drawPaddle() {
    ctx.beginPath();
    ctx.rect(paddle.x, canvas.height - paddle.height,
    paddle.width, paddle.height);
    ctx.fillStyle = paddle.color;
    ctx.fill();
    ctx.closePath();
}```
```

We used the `rect()` method to draw our paddle as a rectangle.

6. Keeping Score

We'll need a variable to keep track of the score:

```
``javascript
```

```
let score = 0;
```

```
``
```

To display it, we'll create a function:

```
``javascript
```

```
function drawScore() {
```

```
    ctx.font = '16px Arial';
```

```
    ctx.fillStyle = 'black';
```

```
    ctx.fillText('Score: ' + score, 8, 20);
```

```
}
```

```
``
```

The `fillText()` method is used to render the score on the canvas.

7. Bringing It All Together

Now, let's create a function to draw everything:

```
``javascript
```

```
function draw() {
```

```
    ctx.clearRect(0, 0, canvas.width, canvas.height);
```

```
    drawBall();
```

```
    drawPaddle();
```

```
    drawScore();
```

```
    requestAnimationFrame(draw);
```

```
}
```

```
``
```

We start by clearing the canvas to ensure no trails are left by the ball. Then we draw our elements. Finally, the `requestAnimationFrame()` recursively calls our `draw()` function, creating an animation loop.

8. Let's Play!

To kick things off, simply call the `draw()` function:

```
``javascript
draw();
``
```

This will initiate our game loop and you'll see the ball, paddle, and score on the canvas.

Summary

In this chapter, we've successfully drawn our ball and paddle using the HTML5 Canvas API and started keeping score. As we progress further, we'll add the ability to move the paddle, bounce the ball, and break the bricks. This is just the beginning of our Breakout game journey!

Creating The Bricks

In this chapter, we'll be focusing on one of the core components of our Breakout game: the bricks. Every game has its obstacles, and in Breakout, the bricks serve that purpose. By using the HTML5 Canvas API, we'll create, style, and position our bricks.

Brick Configuration

Before diving into the code, let's outline the properties our bricks will have:

- Width & Height: The dimensions of each brick.
- Padding: Space between each brick.

- Offset: Initial top and left space to start drawing the bricks.
 - Color: The color of the bricks.
 - Row & Column: Number of rows and columns for our bricks.
-

Setting Up Our Bricks

1. Brick Variables

Before drawing, we need to set our variables:

```
``javascript
const brickWidth = 70;
const brickHeight = 20;
const brickPadding = 10;
const brickOffsetTop = 30;
const brickOffsetLeft = 30;
const brickRowCount = 5;
const brickColumnCount = 3;
````
```

### 2. Brick Array

We'll represent our bricks as an array of objects:

```
``javascript
let bricks = [];
for(let c=0; c<brickColumnCount; c++) {
 bricks[c] = [];
 for(let r=0; r<brickRowCount; r++) {
 bricks[c][r] = { x: 0, y: 0, status: 1 };
 }
}
````
```

Each brick object has `x` and `y` properties for positioning and a `status` property. When `status` is `1`, the brick is visible; when it's `0`, the brick is “broken” or invisible.

Drawing the Bricks

Let's create a function to draw our bricks:

```
``javascript
function drawBricks() {
    for(let c=0; c<brickColumnCount; c++) {
        for(let r=0; r<brickRowCount; r++) {
            if(bricks[c][r].status == 1) {
                let brickX = (c*
(brickWidth+brickPadding))+brickOffsetLeft;
                let brickY = (r*
(brickHeight+brickPadding))+brickOffsetTop;
                bricks[c][r].x = brickX;
                bricks[c][r].y = brickY;
                ctx.beginPath();
                ctx.rect(brickX, brickY, brickWidth, brickHeight);
                ctx.fillStyle = "0095DD";
                ctx.fill();
                ctx.closePath();
            }
        }
    }
}```
```

- First, we loop through our brick columns and rows.
- We only draw bricks with a status of `1`.

- For each brick, we calculate its X and Y position.
 - Using the `ctx` (our canvas context), we draw the brick as a rectangle and fill it with color.
-

Updating Our Game Loop

Now that we have a function to draw bricks, we need to include it in our main game loop so the bricks appear on the screen:

```
“javascript
function draw() {
    // ... other game drawing logic ...
    drawBricks();
    // ... more game drawing logic ...
}
“
```

Conclusion

Our Breakout game now has bricks that serve as obstacles for the player. In the following chapters, we'll look into how our ball will interact with these bricks, removing them when hit, and how to increase the game's difficulty as the player progresses.

Remember, the beauty of game development lies in creativity. You can adjust the properties, colors, and behaviors of bricks to create unique challenges and designs. Happy coding!

Move Paddle

Welcome to Chapter 88, where we will delve into the mechanics of moving the paddle in our Breakout Game. The paddle movement is essential for gameplay, allowing players to deflect the ball and target bricks.

1. Introduction

In the world of gaming, the paddle is the player's only defense against the bouncing ball, preventing it from falling off the screen and enabling the player to aim where they want the ball to go. Using the HTML5 Canvas API, we will be creating this paddle and enabling its movement.

2. Setting Up the Paddle

2.1. Paddle Attributes

First, let's define the paddle's properties. Place these at the top of your script, where you've set up the other game variables:

```
``javascript
let paddleHeight = 10;
let paddleWidth = 75;
let paddleX = (canvas.width - paddleWidth) / 2;
```

- ```
``
```
- `paddleHeight`: The thickness of the paddle.
  - `paddleWidth`: The length of the paddle.
  - `paddleX`: The starting x-coordinate of the paddle. This will place it centered at the bottom of the canvas.

## 3. Drawing the Paddle

Using the canvas context, we can draw the paddle:

```
``javascript
function drawPaddle() {
 ctx.beginPath();
 ctx.rect(paddleX, canvas.height - paddleHeight,
 paddleWidth, paddleHeight);
 ctx.fillStyle = "0095DD";
```

```
 ctx.fill();
 ctx.closePath();
}

```

```

Every time we call `drawPaddle()`, it will render the paddle at its current `paddleX` position.

4. Moving the Paddle

We will allow the player to move the paddle using the left and right arrow keys.

4.1. Event Listeners

Add event listeners to detect key presses:

```
``javascript
document.addEventListener("keydown",
keyDownHandler, false);
document.addEventListener("keyup", keyUpHandler,
false);
``
```

4.2. Handling Key Presses

Now, we'll create our handler functions. When the player presses a key down, `keyDownHandler` will fire. When they release it, `keyUpHandler` will execute:

```
``javascript
let rightPressed = false;
let leftPressed = false;
function keyDownHandler(e) {
  if(e.key == "Right" || e.key == "ArrowRight") {
    rightPressed = true;
  }
  else if(e.key == "Left" || e.key == "ArrowLeft") {
```

```
    leftPressed = true;
}
}

function keyUpHandler(e) {
    if(e.key == "Right" || e.key == "ArrowRight") {
        rightPressed = false;
    }
    else if(e.key == "Left" || e.key == "ArrowLeft") {
        leftPressed = false;
    }
}
``
```

5. Implementing Paddle Movement

Within your game's `draw` or main loop function, implement the paddle's movement:

```
``javascript
let paddleDX = 7; // Paddle's movement speed
if(rightPressed) {
    paddleX += paddleDX;
    if (paddleX + paddleWidth > canvas.width){
        paddleX = canvas.width - paddleWidth;
    }
}
if(leftPressed) {
    paddleX -= paddleDX;
    if (paddleX < 0){
        paddleX = 0;
    }
}
```

```
}
```

```
"
```

Here's a breakdown:

- `paddleDX`: The speed of paddle movement. Adjust to make the paddle move faster or slower.
- The first `if` block checks if the right arrow is pressed and moves the paddle to the right by adding `paddleDX` to its x-coordinate.
- The second `if` block does the same but for the left arrow, subtracting `paddleDX` from the paddle's x-coordinate.
- The inner `if` conditions ensure the paddle doesn't go outside the canvas boundaries.

6. Conclusion

With these steps, the paddle should move smoothly across the canvas, providing the player with a tool to bounce the ball and break the bricks. Remember, adjusting `paddleDX` can change the difficulty level of your game, making it easier or harder for the player.

In the next chapter, we'll dive into moving the ball, which, combined with this paddle movement, will form the core gameplay of our Breakout Game.

Move Ball & Break Bricks

In this chapter, we will dive deep into the Breakout game mechanics by making our ball move and allowing it to break bricks upon collision. We'll use the HTML5 Canvas API to bring our game to life.

Ball Movement

The first thing we need to achieve is to get our ball moving. For this, we'll update the ball's `x` and `y`

position on every frame.

```
``javascript
let ball = {
    x: canvas.width / 2,
    y: canvas.height - 30,
    dx: 2,
    dy: -2,
    radius: 10
};``
```

In the code above, `dx` and `dy` represent the change in `x` and `y` position respectively. If `dx` is positive, the ball moves to the right. If `dy` is negative, the ball moves upwards.

Updating the Ball's Position

To move the ball, we need to update its position on every frame:

```
``javascript
function moveBall() {
    ball.x += ball.dx;
    ball.y += ball.dy;
}```
```

Ball-Brick Collision Detection

To make our game interactive, we'll now make the ball break bricks when they collide.

1. Define the Bricks

First, let's define our bricks:

```

``javascript
const brickRowCount = 5;
const brickColumnCount = 3;
const brickWidth = 75;
const brickHeight = 20;
const brickPadding = 10;
const brickOffsetTop = 30;
const brickOffsetLeft = 30;
const bricks = [];
for(let c=0; c<brickColumnCount; c++) {
    bricks[c] = [];
    for(let r=0; r<brickRowCount; r++) {
        bricks[c][r] = { x: 0, y: 0, status: 1 };
    }
}
``
```

Here, `status: 1` indicates the brick is unbroken. Once broken by the ball, we'll change it to `status: 0`.

2. Collision Detection

For every frame, we need to check if the ball collides with any of the bricks:

```

``javascript
function collisionDetection() {
    for(let c=0; c<brickColumnCount; c++) {
        for(let r=0; r<brickRowCount; r++) {
            let brick = bricks[c][r];
            if(brick.status == 1) {
                if(ball.x > brick.x && ball.x < brick.x +
brickWidth && ball.y > brick.y && ball.y < brick.y +
```

```

brickHeight) {
    ball.dy = -ball.dy;
    brick.status = 0;
}
}
}
}
}

```

```

When a collision is detected, we reverse the `dy` direction of the ball and set the `status` of the brick to `0`, indicating it's been broken.

### 3. Draw Bricks on Canvas

After defining the bricks and detecting collisions, we need to draw the bricks onto the canvas:

```

``javascript
function drawBricks() {
 for(let c=0; c<brickColumnCount; c++) {
 for(let r=0; r<brickRowCount; r++) {
 if(bricks[c][r].status == 1) {
 let brickX = (c*(brickWidth + brickPadding)) +
brickOffsetLeft;
 let brickY = (r*(brickHeight + brickPadding)) +
brickOffsetTop;
 bricks[c][r].x = brickX;
 bricks[c][r].y = brickY;
 ctx.beginPath();
 ctx.rect(brickX, brickY, brickWidth, brickHeight);
 ctx.fillStyle = "0095DD";
 ctx.fill();
 }
 }
 }
}
```

```

```
    ctx.closePath();
}
}
}
}

```

```

---

## Conclusion

By now, you should have a moving ball that can break bricks upon collision. By combining ball movement with brick collision, our Breakout game begins to take shape. In the next chapter, we will focus on handling when the ball hits the game boundaries, the paddle, and resetting the game when the player loses.

# Lose & Reset Game

In this chapter, we'll explore one of the crucial aspects of our Breakout Game: handling the game-over situation and allowing the player to reset the game. Just as in any game, defining and appropriately managing the game's end condition is crucial for an enjoyable player experience.

---

## 1. Understanding the Lose Condition

Before we delve into the code, it's important to understand when our player loses in the Breakout Game. The loss condition is typically when the ball passes the paddle without the paddle hitting it and touches the bottom edge of the canvas.

---

## 2. Setting Up the Lose Condition

First, we need to check in every frame (each time the game updates) if the ball has hit the bottom. Here's how

we can set that up:

```
``javascript
if(ball.y + ball.radius > canvas.height) {
 // Player loses
}
``
```

---

### 3. Resetting the Game State

When the player loses, we should reset the game's state so they can start again if they wish. This involves:

- Resetting the ball's position to the center of the screen.
- Resetting the ball's movement vector (i.e., the direction and speed it's moving).
- Optionally, resetting the score, depending on how you want your game mechanics to work.

Here's a simple function to reset the game state:

```
``javascript
function resetGame() {
 ball.x = canvas.width / 2;
 ball.y = canvas.height - 30;
 ball.dx = 2 * (Math.random() * 2 - 1); // Random
 direction either left or right
 ball.dy = -2;
 score = 0;
}
``
```

---

### 4. Handling Game Over

Now, when the player loses, we'll show a game over message and offer the chance to restart the game:

```
``javascript
if(ball.y + ball.radius > canvas.height) {
 alert('GAME OVER!');
 resetGame();
}
``
```

Note: Using `alert` is a simple way to pause the game and notify the player, but for a polished game, you'd likely want a more integrated game-over screen drawn directly onto the canvas.

---

## 5. Enhancements

There are several enhancements you can add to make the game-over experience more polished:

- **Fade Effects:** Use global alpha and gradually decrease it to create a fade-out effect when the game is over.
- **Custom Messages:** Display custom messages for different score ranges, encouraging the player to try again or congratulating them on a high score.
- **Persistent High Scores:** Using the `localStorage` API, you can store the player's high score on their browser, so they can see their best scores across sessions.

---

## 6. Conclusion

Handling the game over condition and providing a smooth reset experience is crucial for player retention and satisfaction. By clearly defining the end condition, giving immediate feedback, and making it easy to dive back in and play again, you ensure that players remain engaged and motivated to improve.

In the next chapter, we'll explore further enhancements and features that can elevate the gameplay experience. Remember, while the technical execution is essential, it's

the small details in game development that often make the most significant difference in the player's experience.

---

*That's a basic overview of how to handle the game over and reset conditions in a Breakout-style game. You can expand upon this with more advanced game mechanics, visuals, and sounds to create a richer gaming experience.*

## Conclusion

Congratulations on reaching the end of “15 Web Projects With Vanilla JavaScript.” You’ve just walked through a series of fascinating, hands-on projects that pushed your understanding of web development using pure HTML5, CSS, and JavaScript. No frameworks, no libraries—just the raw power of core web technologies.

Reflecting on our journey, we started with the basics, learning the foundations of web development, and then dove deep into understanding how JavaScript interacts with the browser to create dynamic, interactive experiences. From simple form validation, managing local storage, exploring asynchronous programming, manipulating the DOM, to understanding the intricacies of various APIs—every project was an opportunity to harness new skills.

---

### Recap of Key Takeaways:

1. Vanilla JavaScript is Powerful: You don't always need a framework or library to build impressive projects. Understanding the basics well can empower you to create almost anything on the web.
2. HTML5 & CSS are the Backbone: No matter how advanced our JS functionalities were, it all came down to a structured HTML document and styled with CSS. They're the unsung heroes of web development.

3. Hands-on Learning is Effective: Each project was a testament to the power of learning by doing. Theoretical knowledge is essential, but practical application ensures it's cemented in your memory.

4. Debugging is Part of the Process: If you encountered challenges and had to troubleshoot errors—great! That's a crucial part of the developer journey, and each mistake only made you better.

---

### Continuing Your Web Development Journey:

The projects in this book were designed to be both fun and educational, but they also laid the groundwork for more extensive projects. With the tools and knowledge you have now:

- Dive Deeper: Explore other APIs, experiment with more advanced CSS animations, or deepen your JavaScript understanding. The web is vast, and there's always more to learn.
- Incorporate Frameworks: Now that you understand the fundamentals, consider exploring popular frameworks like React, Angular, or Vue. They can help streamline some of the processes you learned here.
- Build Full-stack: Consider expanding your horizons to back-end development. Pairing your front-end skills with server-side programming can allow you to create complete web applications.

Lastly, remember that the tech industry is continuously evolving. To stay relevant, always be curious and keep learning. Engage in communities, contribute to open-source projects, and never shy away from challenges.

Once again, congratulations! Whether this book was a refresher, a new learning experience, or even just a fun exploration of what web development can offer, I hope it served your needs.

Remember, every website, every web application, and every piece of interactive content you've ever enjoyed

online started with someone typing out lines of code. You're now a part of that legacy. Embrace it, keep building, and always keep pushing the boundaries of what you know.